Formal specification based prototyping

Thesis

Formal Specification Based Prototyping

a thesis submitted in partial fulfillment
of the requirements for the degree of Doctor of Philosophy in
Computer Science

Shahram Hekmatpour

Computing Discipline, Mathematics Faculty
Open University

March 1987

to my parents

HIGHER DEGREES OFFICE

LIBRARY AUTHORISATION

STUDENT: SHAHRAM HEKMATPOUR ..... SERIAL NO: HDN67180

DEGREE: PhD in Computer Science .....

TITLE OF THESIS: FORMAL SPECIFICATION-BASED PROTOTYPING .....

.....

.....

I confirm that I am willing that my thesis be made available to readers and may be photocopied, subject to the discretion of the Librarian.

Signed: S. Hekmatpour ..... Date: 15/12/86....

ABSTRACT

Rapid prototyping is an approach to software development which attempts to remedy some of the shortcomings of the linear life cycle model, e.g. its inability to cope with fuzzy requirements and system evolution. This thesis first presents a broad survey of rapid software prototyping. It describes the rationale behind the process, the applications of prototyping, and specific techniques which may be used to achieve them.

We then describe a system, called EPROS, together with its methodology, which supports a number of prototyping techniques in a coherent framework. The system is comprehensive in its approach and covers the prototyping and development of both functional and human-computer interface aspects of software systems. The former is based on the execution of VDM-based formal specification notation META-IV; the latter is based on a textual representation of state transition diagrams. Dialogue development is further supported by a rich set of abstractions which allow interaction concepts to be specified and directly executed rather than implemented.

EPROS is based on a wide spectrum language which supports the main phases of a software development process, namely specification, design, and implementation. Included in this notation is a meta abstraction facility which facilitates its extension by the programmer.

The primary application of EPROS is for evolutionary prototyping, where a system is developed iteratively and gradually from the abstract to the detailed, while it undergoes use and while its capabilities evolve. EPROS copes with all the requirements of evolutionary prototyping, namely rapid development, intermediate deliveries and gradual evolution of the system towards the final product.

The thesis also describes a number of case studies where the presented ideas are put in practice, and which provide data in support of the effectiveness of the described system.

CONTENTS

ILLUSTRATIONS

PUBLICATIONS

The following is a list of author's publications relevant to the research presented here. Some of the material included in this thesis has also appeared in these publications.

PAPERS

- Rapid Software Prototyping, *Oxford Surveys in Information Technology*, Vol. 3 pp. 37-76, 1987 (with D. Ince).
- A Formal Specification Based Prototyping System, in *Proc. BCS/IEE Software Engineering '86 Conference,* Southampton, pp. 317-335, 1986 (with D. Ince).
- Some Applications of Artificial Intelligence in Software Engineering, in *Software Engineering: The Crucial Decade*, Peter Peregrinus, 1986 (with D. Ince and M. Woodman).
- Forms as a Language Facility, *ACM SIGPLAN Notices*, Vol. **21**(9) pp. 42-48,1986 (with D. Ince).
- Software Prototyping: Progress and Prospects, *Journal of Software and Information Technology*, Vol. **1**(1) pp. 8-14, 1987 (with D. Ince).
- A Notation for Specifying Menus, *ACM SIGPLAN Notices*, Vol. **22**(4) pp. 59-62 (1987).
- Experience with Evolutionary Prototyping in a Large Software Project, *ACM SIGSOFT Software Engineering Notes*, Vol. **12**(1) pp. 38-41 (1987).
- The Cluster - a new abstraction in programming, submitted to *IEEE Transactions on Software Engineering*, 1986.
- A Window Manager for UNIX, submitted to *The Computer Journal*, 1986.

BOOKS

- *Software Prototyping, Formal Methods and VDM* (with D. Ince), to appear.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to professor Darrel Ince, who supervised this work; his technical guidance, enthusiasm and constructive suggestions made the completion of this work possible. I am grateful to John Wilson for his kind technical advice on the innards of the UNIX system. Without his guidance I would have been lost in piles of documentation long before doing any useful work. I am also debted to Thomas Carroll for his comments and the time he spent on reading the user manual and a number of other publications.

Chapter 1 INTRODUCTION

No doubt there are programs that are used once and thrown away. No doubt there are even more programs that should be thrown away before ever being used!

- G M Weinberg

1.1 THE LIFE CYCLE MODEL

For the past twenty years or so, software system development has been based on a model, commonly referred to as the software life cycle model [Zelkowitz79, Boehm81, Sommerville82, Shooman82, Fox82]. Though characterized differently by different authors, its overall theme is well-understood and universally acknowledged. The life cycle model leads to a software development strategy which is usually called the phase-oriented, the linear or the traditional strategy.

The life cycle model essentially advocates that software projects should consist of a number of distinct stages; these being: requirements analysis, requirements specification, design, implementation, validation, verification, operation and maintenance. Requirements analysis is concerned with deriving, from the customer, the desired properties and capabilities of a proposed software system. Requirements specification involves stating the system functions and constraints in a precise and unambiguous way. Design is the task of producing, and consequently refining solutions that satisfy the specification. Implementation is the act of realising the design in a programming language which can be executed on the target machine. Validation is the process of checking that a system fulfills its user requirements. Verification has the objective of ensuring that the end product of each of the first four stages matches its input. Operation is the activity of installing and running a completed system in its intended environment. Lastly, maintenance is the process of modifying a system, during its operational lifetime, to correct detected errors, improve performance, and incorporate newly emerging requirements.

The life cycle model was originally derived from the hardware production model of: requirements, fabrication, test, operation and maintenance [Blum82b]. It primarily reflects management concerns in production, such as planning, control, budget expenditure and resource allocation. Its aim is to provide a basis for estimating the correct distribution of

labour and capital over a well-planned period of time by dividing the production process into a number of rationalised phases, each with its own milestones and deliverables.

Central to the model is its linear structure; with exception to validation and verification, all other stages are carried out linearly, i.e. each stage begins only when the previous stage has been completed. The model works very well in hardware production; its appropriateness for software development, however, is becoming increasingly questionable.

1.2 DEFICIENCIES OF THE LIFE CYCLE

Software producers who currently use the life cycle model have to cope with three unpleasant facts. Firstly, the earlier an activity occurs in a project the poorer are the notations used for that activity. Secondly, the earlier an activity occurs in a project the less we understand about the nature of that activity. Finally, the earlier an error is made in a project the more catastrophic the effects of that error. For example, early requirements and specification errors have typically cost a hundred to a thousand times as much as those errors made during implementation [Boehm81] and have lead to a number of multi-million dollar projects being cancelled.

Increasing user dissatisfaction with software since the early nineteen seventies has motivated researchers to pay greater attention to the earlier stages of software development [Ramamoorthy84]. As a result, many requirements analysis and specification techniques have been invented [Davis77, Ross77, Taggart77, Levene82, Lehman85]; some of which are even computerised [Smith76, Teichroew77, Bell77]. At the same time there is a rapidly increasing interest in formal, more mathematical methods of software development which adherents claim lead to more reliable systems which have an increased probability of meeting user needs [Musser79, Davis79, Jones80b, Silverberg81].

Unfortunately, even when a software developer uses modern notations and techniques, success is likely only when the application is both well-understood and supported by previous experience [Bally77, Blum82a, Brittan80]. The current rate of growth in hardware has meant that, each year, large numbers of new applications emerge for which the old knowledge is inadequate. Faster and larger, cheaper memories mean that computers

are being used in novel projects where the relation of the computer to its environment, to human operators, and to other computers has not been researched adequately. Many such projects are based on specifications which are not true reflections of the customer's requirements. This is due to three reasons.

First, there is usually a significant cultural gap between the customer and the developer and the way they communicate [Christensen84]. Consequently, a customer often finds it extremely hard to visualise a system by simply reading a technical system specification document [Gomaa81, Mayr84]. If the customer is unable to visualise such a system then validation during the early part of the project becomes a very error-prone activity. Indeed, the difficulties involved in communication with the user can be a serious barrier to proper development [McCracken82]:

> "The life cycle concept perpetuates our failure so far, as an industry, to build an effective bridge across the communication gap between end-user and system analyst. In many ways it constraints future thinking to fit the mold created in response to failures of the past."

Second, the customer, unfamiliar with information technology, may have produced very vague requirements which could be interpreted arbitrarily by the developer [Brittan80]. Third, empirical evidence [Ackford67, Alter80] suggests that once a user starts employing a computer system, many changes occur in his perception as to what the intended system should do; this obviously invalidates the original requirements. As a result, user requirements are often a moving target, and producing a system that meets them is a risky and error-prone activity.

A further complication is that a software project of considerable size may take many years to complete; during this time the user requirements, as well as the user environment, may change considerably, making the final system even more obsolete [McLean76, Gladden82, Ramamoorthy86]. This is graphically described by Blum [Blum82b]:

> "Development is like talking to a distant star; by the time you receive the answer, you may have forgotten the question."

The life cycle model is strongly based on the assumption that a complete, concise and consistent specification of a proposed system can be produced prior to design and implementation. The validity of this assumption has been challenged and refuted by a

number of authors [Swartout82, McCracken82, Shaw85]. In many cases a complete specification cannot be produced, simply because the user does not really know what he wants [Berrisford79, Parnas86].

Lack of experience in projects where it is almost impossible to construct a precise specification leads to the situation where the customer requirements can be established only when a complete software system has been built and when the system can be examined in a fully concrete form [Blum82a]. For this reason many systems end up being written at least twice. To quote Brooks [Brooks75]:

> "Plan to throw one away; you will, anyhow."

There are numerous examples in the literature of substantial modifications of systems during maintenance because of inadequate requirements analysis. For example, it has been reported [Boehm74] that in some large systems up to 95% of the code had had to be rewritten to meet user requirements. Even more formal, improved techniques and notations for requirements specification are not helpful in this respect, as the transition from the user conceptual model of a system to a specification of the system is an inherently informal process [Leibrandt84].

All evidence, therefore, suggests that the life cycle model has many shortcomings which may have adverse effects on software projects. This is, of course, not to say that this model should be rejected outright. To the contrary, in certain areas, such as embedded software and real time control systems, it is the most rational approach and indeed the best way of controlling the complexity of such projects. However, for the majority of other applications, especially those related to commercial data processing, it is inappropriate and has many deficiencies which are too serious to be ignored. The deficiencies may be summarised as follows:

- It is unable to cope with vague and incomplete user requirements [Brittan80, MacEwen82].
- It discourages feedback to the earlier stages because of the cost escalation problems [Bastani85].
- It cannot predict the effects of introducing a new system into an organisation before the system is complete [Keen81].
- It cannot properly study and take into account the human factors involved in using the

system.

- It introduces a computer system into an organisation suddenly. This is a rather risky approach since users are known to resist significant, sudden social changes [Rzevski84].
- The customer may have to wait for a long time before actually having a system available to him for use. This could have undesirable effects on customer trust and may cause frustration [Gladden82].
- The final product will, at best, reflect the user requirements at the start of the project and not the end. In long projects, these two may differ considerably due to changes in the customer's organisation and practices.
- Once the users start employing the final system and learn more about it, their views and intentions change significantly. Such changes in user perception can by no means be predicted [Clark84].

1.3 THE PROTOTYPING SOLUTION

In the light of the difficulties described above, many researchers have arrived at the conclusion that software development, particularly during its early stages, should be regarded as a *learning process* and practiced as such [Mason83], and that it should actively involve both the developer *and* the customer [Christensen84]. For it to be efficient, it requires close cooperation, and can be successful only when it is based on an actual working system [Somogyi81]. Although customers are not very good at stating what they want from a future software system, they are very proficient at criticizing a current system!

A number of techniques have emerged in recent years that are based on this idea. They are classed under the generic term *rapid prototyping* [Smith82b, Zelkowitz84]. The use of these techniques represents a major change in the way software is produced. They rely on an idea borrowed from other engineering disciplines – that of producing a cheap and simplified prototype version of a system rapidly and early in a project. This prototype becomes a learning device to be used by both the customer and the developer and provides essential feedback during the construction of a system specification. The prototyping approach, when compared to current methods, is so dynamic that the difference can be compared to that between interactive and batch-oriented systems [Naumann82].

Like software testing [Meyer78], the main philosophical issue in prototyping is admission of failure; that we, as human beings, no matter how careful in our development

practices, are likely to make mistakes. Bally [Bally77] puts the idea, appropriately, in the following words:

> "In one sense the prototype strategy is an admission of failure, an admission that there will be circumstances in which, however good our techniques and tools for investigation, analysis and design, we shall not develop the right system at the first attempt. But surely this is only realism based on hard experience, theoretically ideal solutions are often far from satisfactory in a very imperfect world."

One of the objectives of the prototyping approach is to reduce the maintenance effort. There is now considerable evidence [Swanson76, Zelkowitz79, Lientz80, Lientz83] that software maintenance can occupy between 50 to 90% of total project cost during the lifetime of a system. There is increasing empirical evidence [Boehm84] that prototyping can indeed produce more maintainable products.

Overall, the limited results and experience which have been obtained have been very encouraging. For example, in a reported experiment [Boehm84] using prototyping, systems were developed at 40% less cost and 45% less effort than conventional methods. Other researchers have reported even more impressive figures. Scott [Scott78] has described a system which was estimated to cost $350,000 to develop but was accomplished by a prototype that cost less than $35,000. The figures that have been reported have also supported the contention that prototyping shortens the overall development cycle for software [Berrisford79, Mason83, Bonet84].

1.4 THE SCOPE AND LAYOUT OF THIS THESIS

This thesis is a report on the outcome of a research in investigating, developing and integrating rapid prototyping techniques and applying them to the process of software development. It provides a study of the current state of the art in rapid software prototyping , suggests and describes a particular approach, namely the executable specification approach, and combines it with other methods to produce a comprehensive approach for utilising the power of prototyping. The advocated approach is backed up by a fully implemented software development environment, together with working examples and case studies which were performed using the system.

The thesis makes four principal contributions. First, it describes concepts, approaches,

tools and techniques for rapid software prototyping and explores the potential benefits and limitations of its use. It describes the process of prototype development within a systematic framework. Second, it makes a contribution to the integration of diverse areas of mathematical formalism and rapid prototyping. In particular, it promotes the ways in which these two may benefit from one another. Third, it advances the rather under-researched area of wide spectrum languages and programming environments which primarily support the rapid prototyping approach. Finally, it makes a contribution to software abstraction methods by introducing a novel abstraction technique, called cluster, which is of immediate utility in both rapid prototyping and software design.

The earlier chapters of this thesis describe the prototyping approach in a general and critical sense. They provide a background for later chapters which focus on the particular method pursued by this research. Chapter 2 attempts to define prototyping, examine its applications, broadly classify the current approaches, and elaborate on the constituent steps of prototyping projects. Chapter 3 presents a list of technical approaches to rapid prototyping and describes each in some detail.

The specific approach of this work is outlined in chapter 4 which also describes the EPROS prototyping system, its intentions, its scope and the wide spectrum language it is based on – EPROL. Chapters 5, 6 and 7 go into greater depth in describing EPROL and its relevance to specification, design and implementation of functional as well as user interface aspects of software systems within a prototyping framework. Chapter 8 describes clusters and the motivation for their creation.

In addition to many small problems, a relatively large prototyping project was carried out using EPROS to investigate the appropriateness of the methods described in this thesis. This is described in chapter 9. The last chapter examines other work carried out in this area, compares it with the research reported here, and lists a number of future research directions which would be of benefit.

Chapter 2 RAPID SOFTWARE PROTOTYPING

The old order changeth, yielding to new ...
- A Tennyson

2.1 WHAT IS PROTOTYPING?

Prototyping originated from those engineering disciplines which are involved in mass production. There, it refers to a well-established phase in the production process whereby a model is built which exhibits all the intended properties of the final product. Such a model serves the purposes of experimentation and evaluation to guide further development and production. It is important to note that no kind of hardware production is conceivable without going through this phase.

In software engineering the notion of mass production is absent; instead, production refers to the entire process of building the one product. For this reason, the concept of prototyping takes a rather different meaning. Here, most commonly, it refers to the practice of building an early version of the system which does not necessarily reflect all the features of the final system, but rather those which are of interest. In particular, and in contrast to hardware production, we require a prototype to cost very little and take a significantly short time to develop, hence the term rapid prototyping. The purpose, as before however, is to experiment, to learn and to guide further development.

As one would expect with any new term, there is some dispute over the exact meaning of prototyping within the context of software engineering. Some insist that it should be used to refer to a mocked-up initial version of a system which is thrown away after use [Gehani82a, Budde84]. Others suggest that a prototype may become the final system by means of a process of continual improvement [Dodd80]. To avoid confusion, some authors suggest that the term prototype should be used to refer to the *throw-it-away* approach, and the term *evolutionary development* be used when a prototype 'evolves' to become the final system [Gilb81, Patton83].

Other terminologies exist. For example, throw-away prototypes have also been called *scale models* [Weiser82], although it has been argued [Dearnley83] that a model should be regarded as a pictorial representation whereas a prototype is a working system. It has also

been suggested [Gregory84] that a system with a user interface similar to the final product, but incomplete in terms of functionality, should be called a *mock-up* and not a prototype. In contrast to this, the term *bread-board* has been suggested to refer to a system that has a high functionality and no user interface [Botting85].

Other revelant terms used in the literature are: test vehicle, engineering prototype and production prototype [Bally77], heuristic development [Berrisford79], infological simulation [Naumann82], system sculpture [Blum82a], iterative enhancement [Basili75], evolutionary development [Gilb81] and incremental development [Baldwin82].

It is not the intention of this thesis to discuss the merits of all these terms. For our purposes, however, we need to establish what we mean by a prototype. When referring to a prototype, we shall assume the following:

- It is a system that actually works; it is not just an idea or a drawing.
- It will not have a generalised lifetime. At the one end of the spectrum, it may be thrown away immediately after use; and at the other end, it may even become the final system.
- It may serve many different purposes, ranging from requirements analysis to taking the role of the final product.
- For whatever purpose, it must be built quickly and cheaply.
- It is an integral part of an iterative process which also includes modification and evaluation.

We shall, therefore, use the term in a rather broad but, at the same time, controlled sense. Throughout the rest of this thesis, by *prototype* we shall mean a rapid software prototype unless otherwise stated.

2.2 APPLICATIONS OF PROTOTYPING

Prototyping can be applied to various phases of the software life cycle and can also replace some or even all of them. In general, it can be applied in the following areas:

- To aid the task of analysing and specifying user requirements. Here it may have a complementary role, assisting the analyst in finding out actual user requirements. In some cases, the prototype itself may replace the requirements specification document.

- As a complementary tool in software design. For example, to study the feasibility and appropriateness of a system design; to verify novel designs; to contrast and compare the merits of alternative designs; and to demonstrate that a design meets its specification.
- As a tool to resolve uncertainty. For example, to study the effects of, and to cope with, organisational changes due to introduction of new technology; to gradually adapt a computer system to its intended environment; and to decrease the level of risk in introducing automation.
- As an experimental tool, to study the human factors of new computer systems; especially for deriving acceptable human-computer interfaces.
- As a vehicle to support user training in parallel to system development.
- As an economic way of implementing one-shot applications [Smith82b]. These concern problems which may be solved by writing a program and running it only once; after the solution is obtained the program will be of no further use.
- As a complementary tool in software maintenance; especially in situations where due to unstable user requirements heavy maintenance is expected, requiring much of the design to be re-worked.
- As a system development method whereby the prototype evolves to become the final system.

For many technical problems, however, prototyping is not a suitable solution. In such cases, prototyping is likely to have adverse effects, creating more problems than actually solving anything. Example are: space and time efficiency problems, error recovery problems, system security problems, concurrency problems (e.g. deadlocks), hardware interfacing problems, networking problems (e.g. congestion control) and heavy numerical calculations (e.g. solving partial differential equations.)

In general, there are three major areas where prototyping, although possible, is not advisable:

- Embedded software [Zave81].
- Real time control software [Walter84].
- Scientific numerical software [Aggleton86].

Interesting enough, the life cycle model works rather well in these areas and there is usually no need for prototyping. One major area where prototyping could be most valuable is that

which has dominated the software market: commercial data processing. The effectiveness of prototyping here has been demonstrated in many applications such as management information systems [Scott78, Read81, Blum82a], decision support systems [Henderson82], business transaction systems [Dearnley81, Burns86], database applications [Canning81], accounting systems [Earl78], language processors [Zelkowitz80, Kruchten84] and many others.

2.3 CATEGORISING PROTOTYPING

The question of whether a prototype should become the final system is an important one. Even if it is agreed that a prototype will become the final product, other questions, such as how it should be constructed and when it can be accepted as the final product, need to be answered. Because of the importance of the relationship between a prototype and the final system, a classification based on this criterion is appropriate. This is depicted by the following classification which divides the approaches to prototyping into three main categories.

throw-it-away prototyping

This corresponds to the most appropriate use of the term prototype, and is often used for the purpose of requirements identification and clarification [Dearnley81, Kraushaar85]. To stress the relevance of this approach to requirements analysis and specification, it has also been called *specification prototyping* [Keus82] and *specification by example* [Christensen84].

The need for rapid development is the greatest for throw-away prototyping. Since the prototype is to be used for a limited period, quality factors such as efficiency, structure, maintainability, full error handling, and documentation are of little relevance. The prototype may even be implemented on hardware or within an environment other than the one required for the target system. What is important about throw-away prototyping is the process itself and not the product [Floyd84]. The major part of the effort, therefore, should go into the critical evaluation of the prototype rather than its design.

The use of throw-away prototypes, however, is not limited to the specification phase. They may be equally useful in the design phase, as reported in [Dearnley84, Bonet84]. Used in this way, prototypes are often a useful tool for exploring alternative designs and evaluating the appropriateness or feasibility of a new design idea. They are also useful in the testing of a developed system, where they can be used as a comparator that evaluates the correctness of the test results of the system [Weyuker82].

As throw-away prototypes can be easily employed within conventional projects, they do not require any major changes to current software development practices. The cost of throw-away prototyping is highly influenced by the availability of appropriate software tools. Very high level languages have been most commonly used [Zelkowitz80, Gomaa83].

evolutionary prototyping

This approach is in complete contrast to throw-away prototyping [Blum83]; it is in complete antithesis to current software development methods. Proponents of this strategy argue that information systems, once installed, evolve steadily, invalidating their original requirements [Naumann82, Brittan80, Gilb81]. The purpose of the evolutionary approach is to introduce a system into an organisation gradually while allowing it to adapt to the inevitable changes that take place within the organisation as a result of using the system [Rzevski84].

Evolutionary prototyping is by far the most powerful way of coping with change. This approach requires the system to be designed in such a way so that it can cope with change *during* and *after* development. A design practice that does not take the possibility of change into account can lead to severe problems; this is illustrated by the following revealing remark [Alter80]:

> "Systems were strained badly or died as the result of corporative reorganisation ... An old version of a planning model was abandoned as the result of a reorganisation, only to have its basic logic restructured years later ... The conceptual design problem here is building systems that are truely flexible."

In evolutionary prototyping a system grows and evolves gradually [Nosek84, Gilb85]. For this reason, the first prototype usually does not implement the whole

application. Instead, enough development is carried out to enable the customer to carry out one or more tasks completely [Dyer80, Mittermeir82b]. Once more is known about these tasks and how they may affect other tasks, more parts of the system are designed, implemented and integrated with the existing components. This allows a continuous and gradual low-risk development while the system is undergoing use.

Addition and modification are two essential features of evolutionary prototyping and results in new complete deliveries [Gilb81, Patton 83]. Unlike the throw-away approach, the prototype is always installed and used at the customer's site [Rzevski84]. This is of prime importance as the use of a prototype within its actual application environment is the most effective way of performing a comprehensive task analysis.

The primary difference between this approach and conventional software development is that it is highly iterative and dynamic; during each iteration a re-specification, re-design, re-implementation and re-evaluation of the system takes place. As a result, the impact of early errors is far less serious. Furthermore, the initial version of the system is delivered very early in the project and throughout the development process an operational system is always available to the user. This not only supports user training alongside development but also ensures that the final system will not 'surprise' the users when eventually introduced [Hagwood82].

The dynamic nature of this approach, however, may be a considerable challenge to both the developer and the user. Success often depends not only on an effective means of designing an adaptable system but also on a willingness for both sides to open themselves to communication and change for a significant period of time [Floyd84].

At some point in time the final prototype is eventually transformed into the final product. Depending on how well the system design has survived the evolution process the final prototype may serve as the production version or a complete re-design might be necessary to facilitate smoother maintenance. Once again, the availability of appropriate tools is vital. To cut down the re-design effort, a highly modular design which can cope with extension and contraction [Parnas72, Parnas79] should be employed. The success of the evolutionary approach is very much dependent on the ability of the designer to build

flexibility and modifiability into the software from the start [Munson81].

incremental prototyping

Here the system is built incrementally; one section at a time. Incremental and evolutionary prototyping have often been used as synonyms [Baldwin82, Dyer80]. However, there is a significant difference between the two. Incremental prototyping is based on one overall software design [Floyd84] whereas with evolutionary prototyping the design evolves continuously. In incremental prototyping a full scale design is first conducted and then modules are implemented and added in sequence. As with evolutionary prototyping the system grows gradually but in a considerably less dynamic way. Since the incremental approach mostly affects the implementation phase it can be used in conventional software projects [Blum86]. Consequently, it has also been called the *plug-in strategy* [Bally77, Taggart77]. Incremental prototyping provides less scope for adaptation than evolutionary prototyping but has the advantage of being easier to control and manage.

Prior to prototype development the nature of the prototype should be well-understood by both the customer *and* the developer, i.e. whether the prototype should be throw-away, evolutionary or incremental. This point has created considerable confusion in the literature. For example, it has been suggested that it is possible to decide on the nature of a prototype *after* it has been constructed and evaluated [McNurlin81]. This does not seem to be helpful as the design of a prototype is highly influenced by the developer's perception of what it should be used for. For example, because of the significant difference in their expected lifetime, the design of an evolutionary prototype is very different from that of a throw-away prototype [Patton83].

Some authors suggest that prototyping and conventional development methods are complementary rather than alternative approaches to system development [Riddle84, Iivari84]. This is certainly true in the case of the throw-away and incremental approach, but not the evolutionary approach.

2.4 PROTOTYPING ACTIVITIES

To be effective, prototyping should be carried out within a systematic framework. The framework advocated by this thesis consists of four steps. These steps and the way they relate to each other are described below.

the establishment of prototyping objectives

It is essential to establish what a prototype is supposed to be used for and what aspects of a proposed system it should reflect. A clear statement of the lessons that are expected to be learnt from the prototype is also required. This information may be recorded in a document which we may refer to as the prototyping objectives document (POD).

function selection

A prototype usually covers only those aspects of the system from which the required information may be obtained. The selection of the functions to be included in the prototype should be directly influenced by the prototype objectives. Depending on these objectives, prototyping may be carried out *horizontally*, *vertically* or *diagonally* [Floyd84, Mayr84]. Horizontal prototyping involves including all the system functions in a prototype, where each function is considerably simplified and reduced. Vertical prototyping involves including only some of the functions, where each of these is fully realised. Diagonal prototyping is a hybrid of these two. Function selection often boils down to simplifying the original requirements to some extent. However, care should be taken to ensure that the assumed simplifications are both *consistent* and *continuous* [Rich82].

prototype construction

Of great importance is the speed and cost of prototype construction. Fast, low-cost construction is normally achieved by ignoring the normal quality requirements for the final product unless, of course, these are in conflict with the objectives. Throughout construction it must be ensured that everyone is aware of the fact that the main purpose of the prototype is experimentation and learning rather than long-term use.

evaluation

This is the most important step in the prototyping process and must be planned carefully. The users of the system must have already been given proper training and resources should have been made available for evaluation sessions. During evaluation, inconsistencies and shortcomings in the developer's perception of the customer requirements are uncovered. Many features of the system may prove unexpected or inadequate to the user. As evaluation progresses, the customer learns more about the proposed system and his own needs. At the same time, the developer learns about the way the customer conceives the system. The prototype becomes an effective communication medium which enables the two parties to learn about each other, without requiring them to have an in-depth knowledge of each other's fields. The feedback obtained from the evaluation phase must be studied, recorded and used judiciously to improve the prototype.

The prototyping process usually involves a number of evaluation sessions [Naumann82]. After each session, the prototype is modified in the light of the experience gained from its use and then subjected to further evaluation. This process is carried out iteratively until the prototype meets the objectives. The time between the iterations is extremely important. Good, timely feedback is essential for productive learning [Henderson82].

2.5 BENEFITS AND DIFFICULTIES OF PROTOTYPING

The value of the prototyping approach and its suitability for use in software development may be assessed by comparing its advantages against the difficulties it may cause. The advantages may be summarised as follows:

- Prototyping enables one to cope with fuzzy requirements [Bally77].
- A prototype system may be used as a teaching environment. This facilitates user training alongside development. Also, users will not be frustrated while waiting for the target system [Gomaa81].
- A prototype facilitates effective communication between the developer and the user.
- Prototyping gives the user the opportunity to change his mind before committing

himself to the final system [Groner79].

- Prototyping enables the low-risk development of computer systems to be more feasible [Somo81].
- Prototyping enables a computer system to be gradually introduced into an organisation [Hawgood82].
- Prototyping transforms the software development process into a learning process [Gomaa83].
- Prototyping has the effect of increasing the chance that a system will be more maintainable and user-friendly [Somo81].
- Prototyping can reduce the cost and time of development [Dodd80, Naumann82].
- Prototyping encourages users to participate in the development process and improves their morale [Gill82, Earl78].

Prototyping has also its pitfalls and difficulties; these are:

- When carried out in an artificial environment which does not match the final user environment there is a chance that users could miss some of the shortcomings.
- The 'model effect' [Bally77] or 'tunnel vision' [Sol84] might result in inappropriate conclusions being derived from a prototype
- Iteration might not be easily accepted by software designers as it requires the discarding of their own work [Hawgood82, Ramamoorthy86].
- There is a danger that the prototyping process could converge to a set of requirements too quickly, missing some essential points [Henderson82].
- Resource planning and management can be difficult [Alavi84].
- It may be difficult to keep system documentation up-to-date.

Although there is an increasing body of evidence that prototyping has positive implications for the process of software development, a large part of the software community still remain sceptical. Prototyping is not accepted as readily as other engineering disciplines. One reason for this is that software education and training is still strongly based on the conventional model of software development. Another reason is that the prototyping approach still lacks a coherent methodology [Boehm83]. While the former can be solved by updating software courses, the latter can only be solved by further research. The research presented in this thesis is a step towards the latter.

Chapter 3 TECHNIQUES OF PROTOTYPING

A little inaccuracy sometimes saves tons of explanation.

- Saki

In this chapter we describe a number of technical approaches to prototyping. These techniques invariably aim to achieve the same goal – the quick and cheap construction of working prototypes – but vary in the way they go about doing this and the applications for which they may be suitable.

A recent view of software development is that the processing and user interface of a system should be regarded as separate entities and designed as such [Draper85, Hagen85]. This view is adopted here by classifying the technical approaches to prototyping to those that are relevant to prototyping the functional aspects of a system and those that are relevant to user interface prototyping. This classification, however, is not a clean cut; some of the techniques are applicable to both categories. Where that has been the case, we have used a further criterion – the frequency of use in each group.

3.1 FUNCTION PROTOTYPING

An important aspect of any computer system is its functional behaviour, i.e. what it must do. This is normally described by a functional requirements specification document, produced by either the developer or the customer. Waters [Waters79] provides a useful check list of technical facts that must be recorded in such a document. He uses this list to evaluate the completeness of a number of specification languages and concludes that none is even 40% complete. There is also empirical evidence [Bonet84] that once development progresses functional requirements may change and expand considerably. For example, in the case of the project reported in [Bonet84], the requirements expanded by a factor of 5, but were easily controlled by employing a prototyping approach. All this evidence points to the importance of including the functional aspects of a system in a prototype. This section discusses some of the technical approaches to prototyping these aspects.

executable specifications

A promising approach to rapid prototyping is the executable specification approach [McGowan85]. Here, the basic idea is that if a specification language is formal and has operational semantics then it is possible to construct a system that can execute it directly. One attraction of this approach is that it can eliminate the cost of producing a prototype altogether since the specification of a system has to be produced anyway.

Formal specification techniques can be broadly divided into two categories [Liskov75, Claybrook82]. The first category is based on writing a specification as a set of *axioms* [Hoare73, Guttag77, Furtado85]. Axioms may be written as algebraic equations which, when treated as rewrite rules, can specify the operational semantics of the specification. For example, an unbounded stack with three operations of NEW_STACK, PUSH and POP may be specified as:

```
NEW_STACK: --> Stack
PUSH:      Stack, Element --> Stack
POP:       Stack --> (Element | Undefined)
POP(NEW_STACK()) = Undefined
POP(PUSH(stk,elem)) = stk
```

Where the first three lines specify the syntax of operations and the last two lines specify their semantics as axiom. This technique has been employed in the OBJ specification language [Goguen79]. Systems now exist which can translate OBJ specifications into executable code. Similar ideas have been used in the language NPL, its successor HOPE [Burstall80], and also in CLEAR [Burstall81] and SPECINT [Darlington83]. Virtually all these languages allow the axioms to be written as conditional as well as pure equations [Drosten84].

The second category of formal specification techniques is the *abstract model* approach. This is based on specifying the functions of a system in terms of abstract mathematical objects such as sets and functions. The above stack problem, for example, can be specified in an abstract model-oriented method such as VDM as:

```
Stack = Element-list
NEW_STACK: -->
   post(stk,stk') == stk' = <>
PUSH: Element -->
   post(stk,elem,stk') == stk' = <elem> || stk
POP: --> Element
   pre(stk) == stk /= <>
   post(stk,stk',res) == stk' = tl stk & res = hd stk
```

Where a stack is modelled by a list and each operation is specified by predicates on its arguments, result, and the stack. Typical specification languages in this category are described in [Jones80a, Silverberg81, Claybrook82, Sunshine82, Morgan84, Beichter84, Berzins85]. Examples of related executable specification systems are described in [Balzer82, Feather82, Urban82, Henderson84, Belkouche85, Kemmerer85, Lee85].

Henderson and Minkowitz [Henderson86b] provide an excellent comparison of these two categories in the context of executable specifications. They conclude that the differences between these methods are more artificial than real, and illustrate how functional programming could form a suitable basis for both. Similar ideas have also been expressed in [Ardis86].

There are two potential difficulties in making a specification language executable. First, mathematical objects such as infinite sets cannot be represented in finite store and have to be restricted to finite representations. Second, very implicit constructs cannot be easily dealt with and often need to be replaced by more explicit constructs to facilitate execution. Although these problems have no simple solutions, they do not diminish the usefulness of executable specifications. Once a means of execution is available, the work involved in preparing a specification for execution is usually very small [Tavendale85].

Symbolic execution [Cheatham79a, Danenberg82] has also been suggested as a means of both verifying and animating formal specifications. Symbolic execution is a term applied to the execution of programs in a form which produces algebraic rather than numeric values. For example, the fragment of Pascal program:

```
s:= 1;
for j:=1 to 5 do
    s:= s*a[j];
writeln(s);
```

will, when symbolically executed, produce the algebraic expression:

```
a[1]*a[2]*a[3]*a[4]*a[5]
```

rather than a numerical product. This approach has the advantage of addressing the class of all possible implementations for a specification. Discussions of this type of execution to produce prototypes can be found in [Guttag78a, Cohen82, Feather82a]. Unfortunately,

symbolic execution suffers from many problems that are only likely to be solved in the very long term. For example, the symbolic execution of anything but unrealistically small specifications produces an overwhelming amount of symbolic print-out. Consequently, it is unlikely that this technique will play any significant part in software prototyping in the future.

To summarise, even though there are a number of difficult research problems outstanding, there are a number of advantages associated with prototyping by means of specification execution. Apart from being intellectually appealing, this technique ensures that a precise level of documentation is always available to the developer. A specification gradually evolves towards user requirements and, at each stage, a precise description of the system is available rather than being buried in the working detail of a mocked up prototype. Another advantage is the low cost of producing a prototype; little extra work is normally required after a formal specification has been produced.

very high level languages

Very high level languages (VHLL) are programming languages in which it is possible to express complicated operations in a small amount of written program code [Podger79]; they can offer significant gains in increased productivity at the expense of inefficiency in terms of increased running time and storage needs. For this reason they are valuable tools for prototyping. Some of the relevant features of VHLLs are:

- They are interpretive and interactive; a user can interact with such languages in real-time.
- They offer a rich set of objects together with numerous operations on these objects.
- The language notation is short and concise and usually very expressive.
- They are normally supported by powerful software environments and debugging facilities.
- Because of their extensive run-time checks, they are more productive than conventional languages.

One language that has been advocated for prototyping more than any other is APL [Tavolato84]. The basic object in APL is the array and is supported by a large number of

powerful operations. Most APL systems also provide flexible filing systems and a report formatting facility which makes them suitable for prototyping commercial data processing applications. Although APL programs are very concise, they can be quite cryptic and hard to read. Thus, APL is only advisable for throw-away prototyping [McLean76]. A typical use of APL for producing a throw-away prototype for a large commercial system is reported in [Gomaa81].

LISP [Wilensky84] is another VHLL that has been used for rapid prototyping (see for example [Heitmeyer82]). The language itself has a good reputation for very high productivity [Sandewall78]. Also, some very powerful programming environments have been built around LISP and, although primarily conceived as a language for artificial intelligence, it has a number of attractive features making it suitable for rapid prototyping. Amongst these, the uniform treatment of data and programs as lists, a powerful macro facility, and highly interactive features may be named.

PROLOG [Clocksin84] has also been advocated as a rapid prototyping tool [Leibrandt84]. This language is representative of a recent development in programming techniques known as logic programming [Kowalski79] which employs a restricted form of logic to express an algorithm. Currently the language does not enjoy as much popularity as other VHLLs as a medium for prototyping. This is due to poor PROLOG programming environments [Venken84] and partly because PROLOG is still evolving and a number of important technical and language issues have remained unresolved. However, its underlying structure makes it a particularly useful current tool for prototyping database and expert system applications.

Two other VHLLs which have been used for prototyping are SETL and SNOBOL. SETL [Kennedy75] is a programming language which is based on set theory. It has been used in prototyping the first approved compiler for the American Department of Defense language Ada [Kruchten84]. SNOBOL [Griswold71] is a long-established programming language used for manipulating character strings. Zelkowitz [Zelkowitz80] reports on its use in prototyping a language processor.

VHLLs require rather large run-time environments that can consume inordinate amounts

of storage space. This makes them unsuitable for implementing a final product. They also tend to be many times slower than conventional high level languages. However, this does not diminish their utility for rapid prototyping as time and space considerations are often of little concern.

Being real-time and highly interactive, VHLLs enable efficient experimentation with and modification of prototypes; almost a mandatory pre-requisite for prototyping. However, no single VHLL is suitable for all prototyping tasks. Instead a choice should be made by considering which language is suitable for which application domain. For example, if the application in mind is an expert system then APL would be a poor choice while PROLOG or LISP would match the application domain more naturally.

application oriented very high level languages

Application oriented vary high level languages (AHLL) are languages that provide significant savings in implementation time by providing facilities concentrating on a specific application domain such as cost accounting or stock control [Martin82]. These languages are embodied by systems that are either interpretive or program-like. An interpretive system is one in which the user provides a description of an application and the system responds to user requests by performing the desired functions through interpreting the application description; such systems are often known as *application generators*. A program-like system is one in which the user provides a high level program-like description of an application and the system translates it into a program in a conventional programming language; such systems are often known as *program generators* [Lucker86] and the language used is usually referred to as a *fourth generation language* [Read81].

Application generators are highly parameterised and are used to model an application through adjustment of these parameters. The basic idea behind these systems is that if an application domain is well-understood then it is possible to provide systems that can cater for all possible (or at least the most common) functions that would be used in that application domain.

Prywes and Pnueli [Prywes83] describe a program generator which is based on a

non-procedural language [Leavenworth74] called MODEL and is aimed at commercial DP applications. A MODEL program simply consists of a description of data items and a set of equations which describe interrelations between the data items. This description is then translated into a PL/1 or COBOL program. The description is usually compact due to avoidance of input/output detail and the detailed processing that is to occur. Because of this, MODEL programs tend to be 5-10 times shorter than their equivalent COBOL or PL/1 programs. Furthermore, MODEL's comprehensive error checking is a major factor in increased productivity [Tseng86]. The use of MODEL by an accountant, with limited computing background, to generate an accounting system is described in [Cheng84]. Another typical AHLL is HIBOL [Mittermeir82a]. It differs from MODEL, in that it is highly interactive. It allows the interactive definition of business forms and provides facilities for interfacing to a database.

By restricting themselves to small application domains, AHLL systems can achieve high efficiency. As a result, these systems have also been used for producing finished products. In addition, since they facilitate rapid development, they are able to support evolutionary prototyping. The use of such systems for this method of prototyping is detailed in [Canning81]. This reports on the development of a system where the final product contained about 13,000 lines of code most of which was produced by a program generator with the whole development process taking just six weeks.

An attractive advantage of AHLLs is that they can be used by staff with little computing experience. The major disadvantage of AHLL systems is their very limited scope. They are useful for such applications as accounting, payroll, and banking where the application domain is well-understood and where there is a wealth of existing implementation history and expertise [Ramamoorthy84].

functional programming languages

Ever since its early days, computing has been dominated by procedural languages. Such languages allow the programmer to explicitly retrieve data from areas of store, carry out some operation such as addition or multiplication on the data, and then deposit it back into

store again. Procedural languages such as FORTRAN and COBOL have dominated data processing since the nineteen fifties. However, a number of computer scientists have recently pointed out three serious drawbacks with such languages [Backus78, Stoy82]. First, they have become over-complicated. Second, they are unsuitable for implementing software on the multi-processor machines that have been made possible by advances in VLSI technology. That in order to take full advantage of multi-processor architectures some very painstaking and error-prone programming is required. Third, programs expressed in procedural languages are mathematically intractable; it is almost impossible to reason easily about the functionality of large, realistic programs.

As a reaction against the disadvantages outlined above a new generation of functional programming languages [Henderson80, Darlington82] have been designed. The impetus towards their development has been the emergence of new 'fifth generation' multi-processor architectures. Typical functional languages are SASL [Turner79], MIRANDA [Turner85], and ML [Gordon79]. The prime attraction of these languages is their conciseness; functional programs tend to be much smaller and easier to develop than corresponding conventional programs. An example of the conciseness that can be achieved is shown below. It shows a MIRANDA program for taking a finite list of objects and returning the set of all permutations of the list. The corresponding procedural program, expressed in a language such as Pascal,would occupy at least ten lines of code.

```
perms [] = [[]]
perms x = {a:p| a<-x; p<-perms(x--[a])}.
```

Functional programming languages are also a medium for a technique known as *transformational programming* [Darlington76, Darlington81a, Bird84, Barstow85]. This involves a developer producing an extremely concise program for an application which would be very inefficient in terms of memory space and processing time. This program would then be gradually transformed into a working system by the process of replacing inefficient parts by more efficient facilities of the functional language used. This obviously has important implications for evolutionary prototyping.

Functional programming languages are still in their infancy, and many research

questions remain unresolved. Consequently, their scope as a prototyping tool has yet to be explored. However, given promised developments in fifth generation hardware technology over the next decade, functional programming should become an indispensable medium for prototyping.

the tool-set approach

Within the context of software prototyping a tool can be defined as a program that aids the rapid construction of a prototype system. A prototyping tool-set [Glass82] is an environment offering a collection of such tools and a support facility for combining and integrating them quickly and easily.

The most well known tool-set is the UNIX operating system [Bourne83]. Although it was not originally designed for the purpose of prototyping, UNIX offers features that make it suitable for this purpose. The UNIX approach is based on providing a large number of tools [Bell79] that include various language processors, analyser generators, filters, report formatters and many others. The most significant feature of the UNIX tool-set is a uniform and clean common interface. The common interface is called *pipe* and allows the output of one tool to be passed to the input of another tool. Furthermore, the more sophisticated tools, such as LEX and YACC which can quickly generate language processors, have all been interfaced to a common programming language (C).

Prototyping in UNIX often means breaking a problem down into a number of steps where each step is realised by a tool [Kernighan84]. The tools are usually applied successively to data so that the output from one tool becomes the input to another. The high level control which determines the flow of data is obtained through a program known as the *shell* which is a programming language in its own right. In UNIX, the shell acts as glue, joining the tools together with minimal effort. To give an example, consider a program which processes a file of employees, where each employee is represented by a record consisting of his or her name, salary etc., and produces a sorted file of those employees earning more than £10000. It may be implemented as the following shell procedure:

```
cat employees | awk '$2 >= 10000' | sort +0 -1 > high_earnings
```

where the vertical bars are pipes and > writes the output to a file. A number of projects which have used the UNIX tool-set approach are discussed in [Gehani82a, Olsen83, Gray85].

Van Hoeve and Engmann [VanHoeve84] describe a tool-set called TUBA which is specifically designed for the rapid prototyping and development of business application programs. TUBA is built around the programming language Simula-67 [Britwistle73]. It provides facilities for screen formatting and for this purpose it uses a data dictionary to store the pictorial description of objects manipulated and displayed by the system.

reusable software

The relevance of reusable software to rapid prototyping is obvious. If a number of useful modules are available then it is possible to produce a crude, but rapidly constructed, version of a system by joining these modules together. Since the emphasis in prototyping is on ease and speed of construction, reusable modules must have some specific properties. First, and most importantly, they must all have a simple and clean interface [Kernighan78, Meyer82]. Second, they should be highly self-contained; i.e. they should not be dependent on any other module or data structure as far as possible [Parnas72, Hall86]. Third, they must provide some very general functions [Polster86]. Good documentation is, of course, vital. An absolutely minimal documentation standard would insist on a description of each module's interface, function and error conditions.

Reusing old modules is not a new technique; it has been practiced in certain application areas for a very long time. These modules are usually provided in pre-compiled form in a library. The widely-known NAG library of general purpose numerical analysis subroutines is a good example. The domain of applications that have used reusable modules has been very limited. The reason being that not many good general purpose libraries exist. However, the high cost of software development is now providing an impetus to research in this area. This research has included the use of very high level programming languages [Cheng84], the use of a functional programming language to control libraries written in Ada [Goguen84], and the transformation of programs written in one language to another language [Boyle84] or to the same language [Cheatham84]. Recent practical experiences with developing systems

from reusable software are reported in [Lanergan84, Matsumoto84, Litvintchouk84, Polster86].

Since applications vary considerably from developer to developer, it seems reasonable to suggest that each developer should put serious effort into collecting reusable modules [Neighbours84], even though the tight requirements for reusable modules may require a change in a developer's design practice. However, this change should not conflict with good design practices and is, in fact, a strong pre-requisite for good design. A number of criteria for decomposing systems into modules have been advanced [Parnas72, Parnas85]. Much stress is placed on the importance of information hiding and that the design process should start with considering difficult design decisions, especially those that are likely to change with time. Each such decision is then hidden by means of a module. As Parnas demonstrates, this not only results in a clean design but also produces a set of highly independent modules where each has a well-defined function.

Although program code has normally been the medium for writing reusable modules, the ideal medium is a software design notation [Kant81]. The most serious problems that have occurred in employing reusable software have been connected with implementation and programming language details [Balzer83]. A machine-independent software design which has been precisely documented does not suffer from such problems and can normally be implemented quickly on a wide variety of computers and in different languages.

3.2 USER INTERFACE PROTOTYPING

In current interactive systems a large part of the system is devoted to managing human-computer interaction. Sutton and Sprague [Sutton78] report that, on average, about 60% of the program code accounts for the user interface. It should not therefore be surprising that a major part of a software project effort may be expended on the design and implementation of the interface.

User interface design is an inherently difficult task. There are a number of reasons for this. First, specifying a user interface can be very difficult. Written specifications are even less helpful when compared to their use in specifying functionality. There is always a definite

need to be able to visualise the appearance of a system [Lenorovitz77] and this is exactly where written specifications fail. Second, a single system may have a variety of users with considerably different backgrounds [Meurs77, Carey82, Kruesi83]. Attempting to design an interface which is appealing to all users is not a simple task. Third, the complexity of the requirements for a user interface often results in conflicting design goals which necessitate a compromise [Shneiderman79]. It is difficult to detect conflicts on the basis of paper studies and even more difficult to reach a suitable compromise. Lastly, desirable properties of a user interface such as user friendliness and ease of use are highly subjective and are revealed only when a system becomes operational [Tomeski75].

In the past most computer systems were designed with the assumption that the user should adapt to the system. This assumption can no longer be accepted. The majority of current computer users are not data processing professionals and are usually casual users [Benbasat84, Rich84]. It is, therefore, unreasonable to require all users to spend considerable time learning how to use a system, and one has to take novice behaviour into account [Good84].

The traditional methods of software development have been relatively unsuccessful in the design of human-computer interfaces for the following reasons:

- Usually a user interface is not thought of in advance, or not even designed [Mills85]. Most design decisions are left unclear giving the designer the freedom to decide how the user interface should operate. The designer constructs the interface around his own conceptual model which, in most cases, is very distant from the user's conceptual model [Hayes81, Dagwell83].
- The issue of user acceptability [Young81, Foley82] is not dealt with adequately; this inevitably leads to systems which are hard to use.
- The user interface is a major part of the system and is subject to continuous change more than any other part [Munson81]. The need for change is rarely thought of in advance.
- It is now well recognised that the user interface should be designed as a separate entity from the rest of a system [Olsen83, Edmonds81, Green85]. This not only eases maintenance but also simplifies the task of providing a number of interfaces to the same system. This advice is rarely followed. Those parts of the system responsible for human-computer interaction are usually embedded so deeply in the

system that their modification requires changes on a global scale.

Proper design of the user interface is such an important step in system development that many authors believe that it should be the first part of the system to be designed [Hagen80]. The high degree of uncertainty and the possibility of change are good reasons why the design of a user interface should be carried out in an experimental and adaptive manner [Edmonds82] and why it should always consider the user model as an important issue [Green81, Norman83, Draper85]. Unfortunately, the design of such interfaces still remains more an art than a science [Smith82a, Turoff 82]. There are no well-understood procedures that can be followed to guarantee a successful design. Much of what is known is in the form of guidelines [James80, Gaines81]. An obvious problem with using such guidelines is that they are unmeasurable and subjective [Shneiderman79].

The prototyping approach recognises the above difficulties by requiring the design of a user interface to be an iterative process involving a large degree of user participation. This approach allows the designer to derive a conceptual model that is appealing to a majority of users. Actual design of the system only starts when a reliable conceptual model is discovered. There are a number of technical approaches which can be used for prototyping the user interface. They are discussed below.

simulation

One promising approach to the design of human-computer interface is that of simulation [Clark81]. It is a powerful means of studying both user behaviour and the effectiveness of a proposed system, especially when little experience exists of the technology to be used in constructing the interface [Meijer79]. Simulation is especially effective when the problem area is ill-structured [Bosman81].

An interesting use of simulation is outlined in [Gould83]. It describes an experiment in which users were exposed to a 'listening typewriter'. The study was carried out by having an operator and a user in separate rooms each equipped with a VDU terminal. The user would compose his letters by speaking through a microphone. User requests would be intercepted by the operator who would carry them out accordingly, thus giving the impression that the

computer was in control. The aim of the study was to compare user's performance and reactions to a listening typewriter as compared with conventional means of composing letters. The use of simulation allowed the authors to study aspects such as speech mode, size of vocabulary, composition strategy and user experience; most important of all, it enabled them to decide whether an imperfect listening typewriter would be of any utility. This study is important in the sense that it demonstrated that human factors can be studied very effectively through a simple and cheap simulation exercise prior to costly development.

When carrying out a simulation the first task is to derive a simple model of the real system to be developed. This model forms a vehicle for conducting experiments that would otherwise have to be carried out on the real system. The purpose of simulation is to gain insight into the behaviour of a system and also to evaluate techniques behind the operation of a system [Shannon75]. Simulation is a methodology for problem solving and is most effective when the real world experiments are too costly and impractical to perform. Some authors consider prototyping as a specific instance of simulation [Sol84].

formal grammars

A useful mathematical tool for the specification and design of human-computer interaction are formal grammars. These are notations used to describe the syntactic structure of various languages. The most commonly used notation is the Backus-Naur Form (BNF) which was originally designed for the specification of the syntax of programming languages [Naur63].

The specification of a human-computer dialogue consists of two parts; the first part is the specification of the user input; the second part is the specification of the system's response to that input. Using BNF, one can easily specify user input formally and concisely [Shneiderman82]. The specification of a system's response to user input is not possible without extensions to BNF. Such an extension will introduce semantic actions into a BNF description. These actions check the validity of the user input and perform the required requests. For example, a simple mailing system with a single command for sending documents to users may be specified as:

```
<mail> ::= 'send' document 'to' user [send_mail($name1,$name2)]
<document> ::= {'A'..'Z'}+ [$name1 = match]
<user> ::= {'A'..'Z'}+ [$name2 = match]
```

where the parts enclosed in square brackets represent the semantic actions. This approach has been used in a tool which takes a BNF description of graphical input devices and produces a prototype user interface [Hanau80]. A similar, but more flexible approach, is described in [Olsen83].

If the user interface is based on a simple command language then compiler generator tools can be used to prototype the user interface. Such tools have, in the past, enabled developers to rapidly produce translators for programming languages from a BNF description of a language. One such tool which has proved useful in user interface design is the UNIX-based YACC compiler generator [Johnson75].

A different and more ambitious approach to the use of formal grammars for dialogue design involves what is known as the command language grammar (CLG) [Moran81] which describes a user interface at four levels; these being task, semantic, syntactic and interaction levels. CLG, although important as an attempt to extend the use of formal grammars, does not seem to be immediately suitable for prototyping purposes. It produces very long and detailed specifications that are often too complicated to comprehend. Furthermore, no automated tools are available to support its use. CLG, however, is a useful conceptual framework for the specification and design of dialogue systems [Davis83, Browne86].

An interesting use of formal grammars for prototyping has been suggested by Reisner [Reisner81] who used formal grammars as a predictive tool to make a pre-development comparison of alternative designs. She predicted that certain properties of the BNF description of a user interface determine the complexity of the interface. To substantiate her claims she performed an experiment that demonstrated the correlation of empirical results of user performance with her predictions. Two similar approaches to user interface evaluation using formal grammars are described in [Blesser82, Wang70].

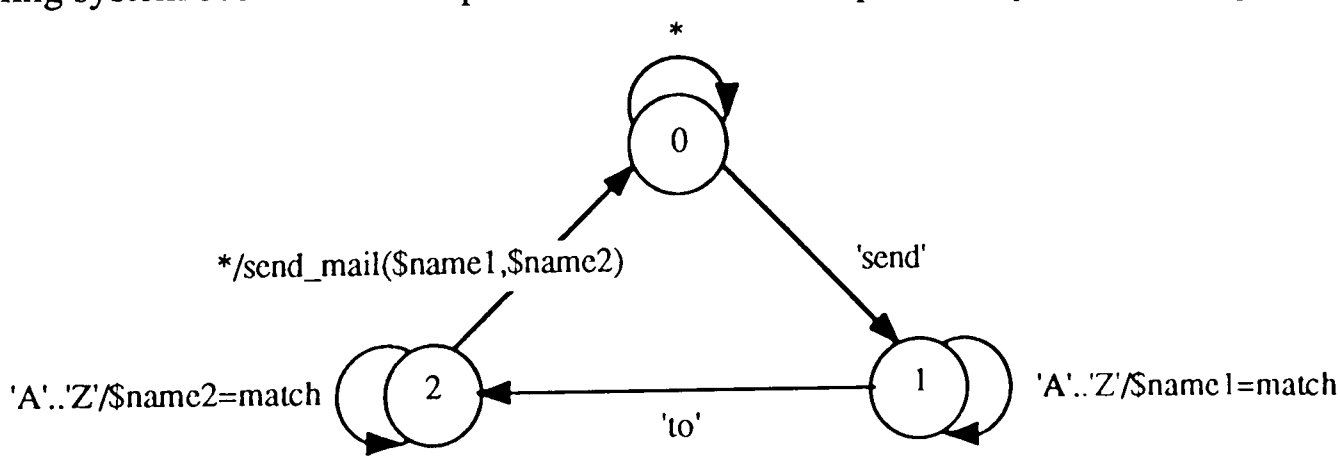
Formal grammars are by no means the ideal tool for dialogue design and prototyping. They have a number of problems [Jacob83]. Firstly, for any serious dialogue, the BNF description can become very complicated and incomprehensible. As a result, it may be very

difficult to decide what event might occur after a series of user actions and vice-versa. Secondly, BNF is particularly weak in describing error cases and help messages. Such messages must occur at very specific points in the dialogue and their inclusion often requires adding further complicated rules to a BNF description which do not seem to correspond to any reasonable concept.

state transition diagrams

The use of state transition diagrams (STD) for dialogue specification and design was first proposed by Parnas [Parnas69]. The concept has also been used for specifying the functional requirements of computer systems [Casey82]. An STD is a directed graph consisting of nodes and edges. Each node is usually represented by a circle and depicts a state of the dialogue. Nodes are connected by edges representing transitions between states. With each edge an input stream may be associated indicating that a transition between states will occur if the user input matches the specified input stream.

STDs of this type have been used in the design of lexical analysers, parsers and compiler generators [Conway63, Johnson68]. Like formal grammars, in order to be useful for dialogue design, some extensions to the STD notation are necessary. Two extensions are usually provided [Casey82, Kieras83, Wasserman85]. The first extension allows each edge to be labelled with an output message. This message is sent to the user when the associated transition takes place. The second extension incorporates semantic actions into an STD. These actions are again associated with edges and are invoked by transitions. For example, the mailing system described in the previous section can be specified by the following STD:



where a slash is used as a separator between user input and corresponding computer action.

A number of tools have been constructed that convert an STD specification of a user

interface into a prototype of the user interface [Wasserman79, Wasserman82a, Wasserman82b, Jacob83]. In all these tools a linear textual notation is used for specifying STDs. The tools process the specification of a command driven user interface, expressed in this notation, and generate a finite state automaton which acts as a prototype of the user interface.

An important characteristic of STDs is that they make the state of a dialogue explicit and hence more readable; with formal grammars, this information is always hidden and usually very difficult to extract. This feature of STDs is very important from the point of view of the staff carrying out prototyping [Norman83]. An STD specification is also an exceptionally useful aid in the design of error/help facilities [Feycock77]. In an interesting study [Guest82] two dialogue design systems were compared. One system, SYNICS [Edmonds84], used the formal grammar approach while the other was based on state transition diagrams. SYNICS was rejected by almost all the staff who used it; the reaction to the other system, however, was positive and was even used productively by non-computer experts.

other formal methods

There are a number of other formal methods which have been applied to the specification and, in some cases, prototyping of user interfaces. One approach reported in [Hopgood80] uses production systems as a basis for specifying human-computer interaction. Production systems are extensively used in expert systems and are based on situation-action or if-then rules [Winston81]. The use of production systems in dialogue design involves producing a knowledge base of if-then rules, where each rule associates a predicate over user input (and possibly systems states) to a system action in response to that input.

Although similar in some sense to state transition diagrams, production systems are distinguished by the fact that they avoid specifying I/O order. Interesting enough however, an STD can always be mapped to an equivalent production system easily.

Formal functional specification notations have also been applied to dialogue specification. Examples of these are given in [Feather82b, Sufrin82, Sufrin86, Cook86,

Meandzija86]. Chi [Chi85] provides an interesting evaluation of the use of four formal notations for dialogue specification which includes both algebraic and model-oriented methods. He demonstrates that, while these notations are capable of specifying interaction, their use is difficult and time-consuming. This is not surprising, as these notations were not originally invented for the purpose of dialogue specification, and fall short of many useful dialogue-oriented features. Indeed, what they lack most is a suitable underlying model for specifying interaction.

One such model is described in [Alexander86] and is an extension to a current formal specification and prototyping notation, called ME-TOO [Henderson86]. This model is based on the notions of dialogue events and finite state machines, and uses a LISP-like read-eval-print concept to model interaction. It retains the functionality of ME-TOO and, having been based on an executable notation, it is capable of prototyping human-computer interaction.

Another method is described in [Silbert86] and is based on the object-oriented programming paradigm. In this model a user interface is designed as a network of objects of pre-defined classes which depict different views of the dialogue and which communicate to one another by passing messages. The model is primarily intended for graphical user interfaces but is general enough to be applicable to other applications as well.

screen generators and tools

The appearance of the screen display is usually of great importance to a user. The traditional methods of screen design usually rely on producing paper drawings of screens. There are two difficulties with this approach. First, the drawings can take considerable time and effort to produce. Second, experience has shown that what seems to be acceptable on paper appears very different when displayed on a VDU screen. Software developers now recognise that the best way to reach an agreement on screen layout is by actually producing them on a VDU and carrying out a repeated process of modification until the user agrees with the presentation. However, programming such screen displays is a time-consuming and expensive task and can only be economically carried out by means of prototyping tools.

Screen prototyping tools fall into two categories. The first category is based on providing a high level notation for screen definition. This is implemented either by means of a processor which converts screen definitions to a prototype version of the screen display [Christensen84], or as a package of library routines accessible from a programming language [Dixon85, Sale85, Kenneth81].

The second category makes use of sophisticated screen editors to produce the screen layouts interactively [Mittermeir82a], where each time a screen is produced it can be stored in a database and subsequently re-displayed. Both approaches allow rapid generation of a *scenario* of the application user interface. A scenario is a way of presenting to the user the sequence of events he or she would experience while performing some task and is more concerned with the presentation than the actual processing behind it.

The use of scenarios for the design of interactive systems has been advocated as the most eloquent way to design a human-computer interface [Hooper82, Mason83]. Scenarios usually contain little or no application logic, so the sequence of events occurs in a predetermined, fixed order. This, nevertheless, is a very useful concept which allows the user to experience a system without the developer committing much resources to implementation.

Mason and Carey [Mason83] have employed these ideas in a systematic way. They have devised a technique known as the *architecture-based* methodology. It takes its name from the similarity of the approach to the way buildings are developed; the technique places great importance on the external view of a system. The designer starts with an external view of the system and works inwards from this. During this process the designer has the responsibility of ensuring that the system appearance is both acceptable and understandable to the user. The methodology is supported by a tool called ACT/1 which rapidly produces scenario prototypes of systems.

In a way, the architecture-based methodology is the reverse of conventional approaches to system development where the system grows from inside outwards with its appearance becoming known only when it is fully constructed. The most significant advantage of this methodology is that it ensures that the system appearance is acceptable to

the user during the whole of the development process. A limitation of this methodology is that it is only suitable for producing interactive information systems. In these systems the user interface dominates the entire system and its quality accounts for the quality of the system as a whole. The architecture-based approach is representative of a number of recent approaches which argue that system development should start with the user interface. Other approaches which make use of tools to aid the construction of user interfaces are described in [Buxton83, Aaram84, VanHoeve84].

language supported facilities

Another way of prototyping user interfaces is via facilities built into a programming language [Shaw83]. These facilities have the potential of eliminating the need for dealing with the very low-level detail commonly found in programming human-computer interfaces.

Almost all current programming languages were designed with an emphasis on batch processing rather than interactive computing [Shaw83]. This is evident from the type of input/output facilities provided by them; these facilities are usually limited to reading and displaying strings and numbers. Modern interactive systems rely on much more flexible and powerful concepts of interaction (e.g. windows) [Hagen85]. Therefore, it is not surprising that much of the design and programming effort in user interface construction is expended on implementing these facilities by employing painstaking, laborious and error-prone low-level programming. Early work in this area has been centred around very high level languages. Examples include the use of LISP for prototyping command languages [Levine80] and the report generation facilities of APL [Tavolato84].

There are four types of facilities which are increasingly being used in modern interactive systems; these are electronic forms, menus, overlapping windows and icons. Suitable extensions to programming languages would allow the use of these facilities to be 'specified' rather than programmed [VanWyk82, Mallgren82].

The specification and design of electronic forms using language supported facilities is extensively described in [Gehani82b, Gehani83, Yao84, Tsichritzis82]. Language facilities for specifying and prototyping icons and menus are discussed in [Brown82, Gittins84,

Lafuente78]. Use of windows is detailed in [Teitelman79, Rowe83]. The provision of programming language constructs to support abstract input/output tools is discussed in [Bos78, Bos83].

3.3 DISCUSSION

It would be useful to compare the techniques described above in terms of their potential application domains and usefulness. This is summarised in figure 3.1. Examination of this figure leads us to the conclusion that none of the techniques can, on its own, be regarded as a complete and comprehensive prototyping tool. Each technique, while capable of capturing some aspects of an application, falls short of being applicable to others. Even the ones which have been classified as general have their own problems. In the reusable software approach, for example, no matter how many reusable modules we have at our disposal, moving to a new application will always require the development of additional unforeseen modules.

TECHNIQUE	DOMAIN	ADVANTAGE	DISADVANTAGE
executable specs.	functionality	concise & productive	not all specs. are executable
VHLL	language-dependent	productive	often cryptic
AHLL	very restricted	very productive	very application dependent
functional PL	functionality	concise	often inflexible
tool-sets	tool dependent	very productive	incoherent
reusable software	general	very productive	initially expensive
simulation	general	early application	no general support tools
formal grammars	certain interactions	concise	inflexible
STD	interaction	graphical	textual notation often cryptic
screen generators	mostly static dialogues	productive	inflexible
language facilities	language-dependent	concise & productive	restricted utility

FIGURE 3.1 A comparison of prototyping techniques.

Previous researchers have concentrated on devising systems that each support only one of the above techniques (see for example [Goguen79, Jacob83, Mason83, Olsen83, Prywes83, Shaw83, Cheng84, Turner85].) This in turn has limited the utility of such systems for prototyping. The incompleteness of individual techniques and their highly different properties suggest that a *combination* of some of these techniques may be required in order to produce a powerful and general prototyping tool. This in fact is one of the major

issues that we shall be exploring in this thesis. Our interest, therefore, will lie in integrating a number of prototyping techniques so that they will compensate for each other's shortcomings.

The next chapter will describe the combination of techniques that we have adopted and a system that implements and integrates these techniques within a coherent framework. The combination may seem rather arbitrary and is obviously one of many possibilities. We shall show, however, that it is an effective one and that it can accommodate all prototyping approaches described previously.

Chapter 4 THE EPROS PROTOTYPING SYSTEM

In this chapter we give an overview of our approach to prototyping and its application to system development. The approach and its methodology are supported by a development and prototyping environment called EPROS. In EPROS a system is developed in a top-down manner, from the abstract to the detailed. Progress is iterative and cyclic where each cycle produces a self-contained description of the system. This description, no matter how abstract or how detailed, is always executable and is automatically converted into a working prototype.

4.1 THE APPROACH AND ITS SCOPE

The EPROS approach is based on utilizing and integrating four technical approaches to prototyping (see chapter 3); these are:

- Executable specifications
- State transition diagrams
- Language supported facilities
- Reusable software

The functional requirements of a system are formally specified in META-IV [Jones80a, Jones86]. EPROS automatically translates such specifications into working prototypes. The user interface of a system is formally specified using state transition diagrams [Denert77]. EPROS provides a textual notation for describing these diagrams which is readily executable. User interface development and prototyping is further backed up by language supported facilities which have been especially designed to simplify the task of constructing user interfaces. Language supported facilities can be readily extended by the programmer through a facility called cluster which is also the main tool for reusable software development.

EPROS supports the three main approaches to prototyping; namely, the throw-away, the incremental and the evolutionary approach. When used for throw-away prototyping, a system is first formally specified and then automatically converted into a prototype. Next, the prototype is evaluated by the user, whose feedback is used to improve the prototype. Any

changes to the prototype are carried out by modifying the specification and regenerating a new prototype. This process is repeated until the prototype converges to a stable set of user requirements, at which time the prototype is discarded and the final system description is used for initiating a separate development process.

When used for incremental prototyping, an overall specification of the system is first produced (possibly using the throw-away approach.) This specification is refined to generate a design which is then frozen. A small subset of the design is selected as the first increment; this is fully developed and handed over to the customer. The rest of the design is broken down into subsequent increments which are developed similarly and handed over to the customer one by one. User feedback obtained during this process is used to improve the increments. The architecture of the system, however, will remain intact; any requested changes will be restricted to the implementation of the increments.

EPROS is primarily intended to be used for the evolutionary approach. Evolutionary prototyping has three important requirements: fast iterations, intermediate deliveries, and gradual evolution of prototypes towards the final product. The executable specification features of the system cope with the first two requirements. The system also provides extensive facilities for the design and implementation of software systems; these support the last requirement of the evolutionary approach. Because of this comprehensive support, the entire development takes place within the system and is expressed in one notation, i.e. EPROL.

EPROS relies on the use of formal methods and notations for two reasons. The first reason is the potential of these methods for the automatic and fast generation of prototypes. The second reason is the power of these methods in producing clean and flexible designs [Jones77, Musser79, Feather82b, Sufrin82, Morgan84, Berzins85, Minkowitz86, Weber86, Ford86]. This is highly crucial and indispensable for evolutionary prototyping as, without a good design, modifications and extensions become totally impractical. VDM was chosen as the underlying formal method since it is a well-developed methodology and has been used successfully in the development of many non-trivial systems [Hansal76, Cottam84, Minkowitz86, Bloomfield86].

4.2 THE DEVELOPMENT PROCEDURE

Figure 4.1 shows a schematic view of the evolutionary prototyping procedure of EPROS. Development always starts with an informal specification of user requirements, which may be vague, incomplete and unstable. After a preliminary study of the requirements a formal specification is produced. The first specification may consider only functional requirements, or only those related to the user interface, or both. Usually, however, one starts with the functional requirements, in which case, they provide a backbone and context for formulating the user interface requirements.

The formal specification is then converted into a working prototype and is evaluated by the user. After a few iterations, which may result in changes and/or extensions to the specification, the specification is refined. Each refinement produces a prototype for evaluation and more iteration. At some stage, the functional part of the system and the dialogue part are integrated. Integration can also take place before the refinement of the specification. The issue of when to integrate is really application dependent and is influenced by the way the project progresses. However, before integration starts, the user must be fully satisfied with the exhibited behaviour of the system.

The result of integration is a further prototype. Evaluation of this prototype will reveal whether a loop back to a previous stage is necessary or not. Once the system is integrated, it is repeatedly refined. Each refinement produces a complete delivery in form of a prototype. During the refinement process, abstract constructs in the system are replaced by more concrete ones. This process continues until the system is in its most concrete form and the last prototype may be tuned and released as the final system.

The development process can also be complemented with formal verification. This is not shown in figure 4.1. Verification can be applied to the specification and refinement steps. Experience with the methodology, however, suggests that verification is usually cost-effective only when it is applied to the top level specification, after it has been evaluated and agreed upon. The reason for this is that top level specifications are very abstract and, therefore, easy to verify; but the more the system is refined the harder verification becomes.

Also, errors is the top level specification are much more costly to correct than those in the refinements.

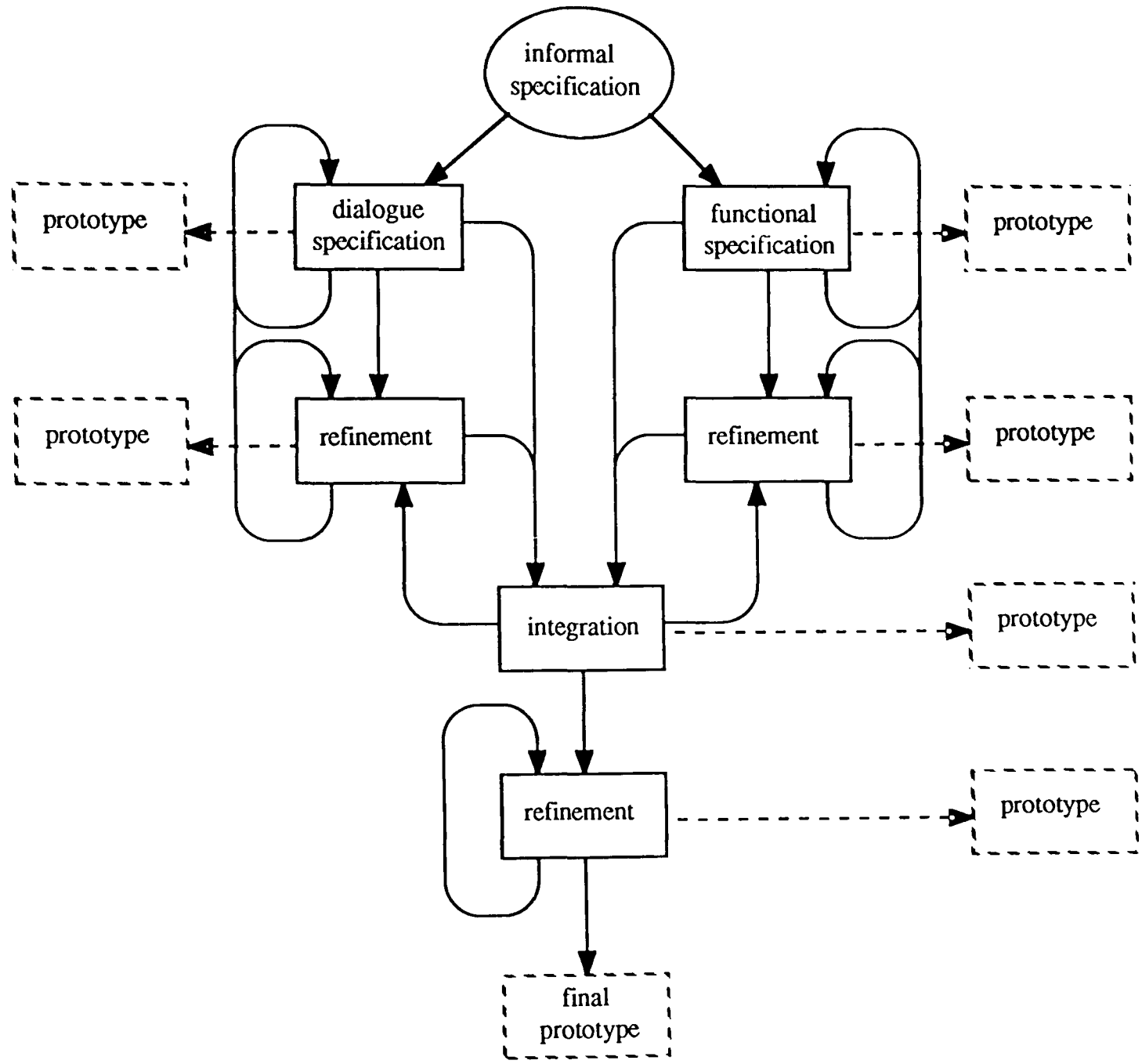


FIGURE 4.1 The evolutionary prototyping procedure of EPROS.

4.3 THE EPROL WIDE SPECTRUM LANGUAGE

EPROS is based on a wide spectrum language called EPROL which supports the formal specification, design and implementation of software system. EPROL is both a prototyping and a development language. It provides facilities for dealing with functional and dialogue aspects of a system, and is fully executable. Various facilities of EPROL are briefly described below. The syntax of EPROL is formally specified in appendix A. For a much more comprehensive description of the system and its language see the user manual

[Hekmatpour86].

functional specification notation

The functional specification notation of EPROL is based on META-IV - the formal specification notation of VDM [Bjorner78]. Specifications are written using mathematical notations and objects such as predicate calculus, sets, lists, mappings, abstract syntax, applicative combinators and pure functions. Side-effects are specified by pre- and post-conditions over a class of states, in an abstraction called an operation. The main notation for specifying functionality is the abstract data type notation; it is used for specifying new data types, i.e. those which are not directly available in the language [Rowe83].

EPROL provides a number of extensions to the META-IV notation. Amongst these are: polymorphic types, operator mapping and operator distribution.

dialogue specification notation

Dialogues, in EPROL, are specified by state transition diagrams. The notation for STDs is based on the graphical notation of Denert [Denert77], which distinguishes between three kinds of dialogue states. These are simple states, complex states and interaction points. A simple state refers to a computer action involving no interaction with the user. A complex state is an abstraction of an entire STD and may involve interaction with the user. An interaction point is where actual interaction with the user takes place. The notation of complex states allows dialogue specifications to be modularised in much the same way functional specifications are.

design notation

The term design, in EPROL, refers to the refinement and modularisation of a software system. In addition to abstract data types, four other kinds of modules are available for this purpose:

- *Functions* which roughly correspond to functions and procedures in modern programming languages. These are by nature imperative and can have a hierarchical

structure.

- *Dialogues* which correspond to complex states in a dialogue specification. These are again imperative and can have a hierarchical structure.
- *Forms* which are used for defining electronic forms as abstract data types. These are non-hierarchical and object-oriented.
- *Clusters* which provide a powerful mechanism for extending the base language, introducing new abstractions and designing reusable software modules. Clusters have syntax driven interfaces and can also have a hierarchical structure.

The rules for the use of the above modules in hierarchical design are depicted by figure 4.2. Directed arrows should be read as 'may contain.'

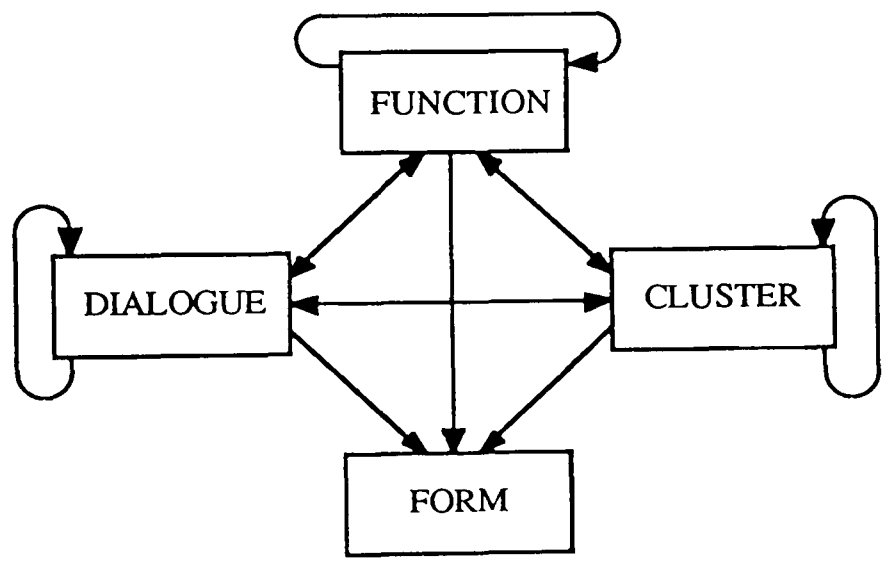


FIGURE 4.2 Module containment in EPROL.

implementation notation

The implementation notation is based on a hybrid of C and Pascal, and is strongly typed and structured. Notably, all the constructs of META-IV are also supported by the implementation notation. So, the notation can be purely applicative, purely imperative, or a mixture of the two.

4.4 THE ARCHITECTURE OF THE SYSTEM

The architecture of EPROS is shown in figure 4.3. The system is partitioned into 11 independent components. Central to the system is the EPROL compiler which implements the EPROL language. The compiler is itself divided into three major partitions which in turn cover

the functional specification, the implementation, and the dialogue notations. These components are further modularised in a way that reflects the modularisation of the notation. This open architecture has the advantage that the notation can be upgraded during the lifetime of the system with minimum amount of effort. Appendix B describes an example which illustrates the use of the EPROL compiler.

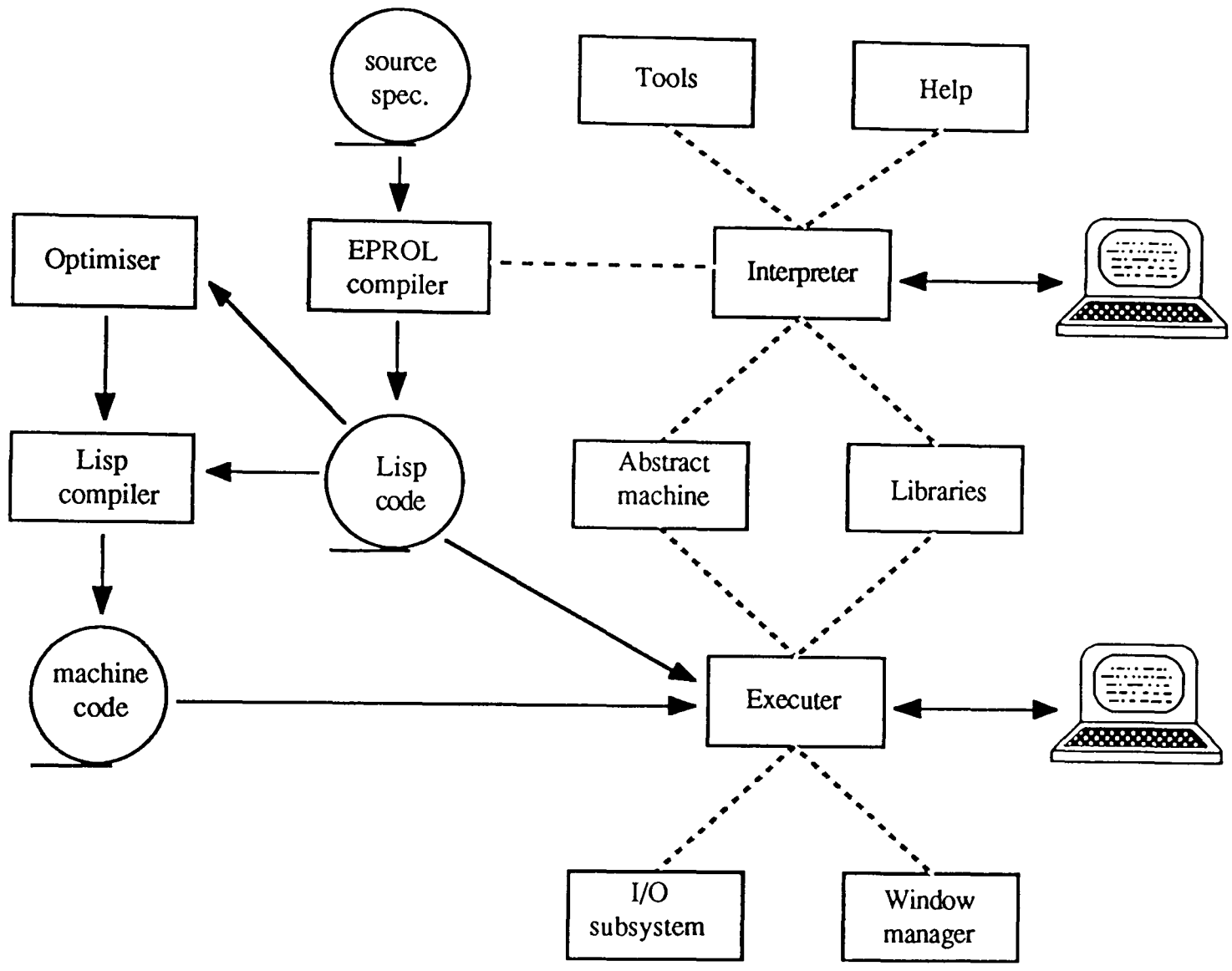


FIGURE 4.3 The architecture of EPROS.

The second major part of the system is the interpreter, which sits directly on top of the compiler. The interpreter allows direct, interactive access to EPROL. The style of interaction is very much like a LISP environment, with the following exceptions:

- Two interaction modes are provided. The first, the expression mode, restricts the user to the functional specification notation. Computations causing side-effects are strictly prohibited in this mode. Also, the result of any interactively typed expression is immediately displayed (as in LISP.) The second interaction mode, the statement mode, allows any form of computation. The display of end-results in this mode is intentionally avoided; these may be optionally displayed using explicit I/O statements.

- The interpreter provides an interface to the symbol table of EPROL. This means that the user can find out what is currently defined. The interface also allows the user to remove unwanted definitions. Objects can be referred to either individually (by specifying an object's name), or collectively (by specifying a category, e.g. FUNCTIONs, CLUSTERs, etc.) The notation used for displaying objects is that of EPROL, and is handled by a dedicated pretty-printer.

Using the interpreter, the user can interact with an already compiled EPROL file, or alternatively, create his own definitions.

The interpreter has direct interface to four other components of the system: the help subsystem, the tools, the abstract machine, and the libraries. The help subsystem provides interactive on-line help on a variety of topics, which include system commands, interpreter commands, and all syntactic components of the EPROL notation (e.g. operators.)

The tools part consists of a set of pre-developed tools. These currently include a cross-reference tool for EPROL, a highlighter for the neat display of EPROL files on the screen, and a pretty-printer for pretty printing META-IV objects. In addition, new tools can be included with considerable ease, without disturbing the overall system.

The abstract machine part consists of a set of compact and highly optimised routines which implement the abstract objects of META-IV (i.e. sets, lists, mappings and trees.) The libraries part consists of a set of predefined standard libraries for EPROL. (See appendix C for a brief description of each library.)

Both the abstract machine and the libraries are also shared by the executer component. The executer has the role of executing finished products (i.e. ones which have gone through design iteration.) This component can be run quite independent of the rest of the environment to achieve greater efficiency by avoiding the overhead of unnecessary components.

The executer is, in turn, interfaced to the I/O subsystem and the window manager, which collectively support the dialogue mechanisms of EPROL. Both these are based on an internal notation which is hidden away from the user. This has the obvious advantage that these components can be changed (possibly in the event of porting the system to a new hardware configuration) without actually affecting the EPROL notation.

The remaining two components of the system (the optimiser and the LISP compiler)

deal with the code generated by the EPROL compiler. The optimiser performs some straightforward improvements on the intermediate LISP code. The LISP compiler is a customised version of the standard Franz LISP compiler in the UNIX environment (i.e. Liszt.) It simply translates the intermediate LISP code into machine code.

EPROS is implemented as two monolithic programs. The first program (eps) is the entire environment and includes all the components shown in figure 4.3. The second program (epx) consists of the executor, the I/O subsystem, and the window manager. It is intended to be used for running complete systems only. The overall system has the following features:

- Compilation speed of approximately 1000 lines of EPROL source per minute.
- Full error detection, reporting and recovery.
- Separate compilation.
- Compiler switches.
- Compiler directives.
- Optional object code optimisation.
- Various useful libraries.
- An extensive interactive synopsis and help facility.
- Various useful tools.
- An interface to the UNIX operating system, allowing interactive execution of UNIX commands from within the environment.

EPROS was developed and runs on a VAX-11/750 computer under Berkeley UNIX 4.2. It consists of 412 modules and occupies just under 20,000 lines of code. Two thirds of the system was written in Franz LISP; the remaining third, which contains the main bottlenecks of the system, was written in C for the sake of efficiency.

The system itself was developed using an evolutionary prototyping approach which consisted of 12 development cycles. Each cycle lasted about 6 weeks, with major reviews at the end of every second cycle. The approach proved very effective; although currently a prototype, the system matches the quality of a finished product very closely.

Chapter 5 FUNCTIONAL SPECIFICATION

Abstraction is the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases.
- E W Dijkstra

The specification of a software system is divided into two parts. The first part specifies the functional requirements of the system and is described in this chapter. The second part specifies the user interface requirements of the system and is described in chapter 7. The notations and methods presented also cover the design stage where decisions about how the requirements are to be realised are made.

As stated earlier, the functional specification notation of EPROL is largely based on VDM, a brief outline of which is given below.

5.1 THE VIENNA DEVELOPMENT METHOD

VDM is a constructive or abstract model-oriented formal specification and design method based on discrete mathematics [Jones80, Bjorner82]. The formal specification language of VDM is known as META-IV and is extensively described in [Bjorner78]. VDM only considers the functional specification and development of software systems. Other aspects, such as the user interface, have to be developed using other notations and methodologies.

Very briefly, in VDM, a system is developed according to the following steps:

specify the system formally.
prove that the specification is consistent.
do
    refine and decompose the specification (realisation).
    prove that the realisation satisfies the previous specification.
until the realisation is as concrete as a program.
revise the above steps.

In VDM, a specification is written as a constructive specification of a data type, by defining a class of objects and a set of operations to act upon these objects while preserving their essential properties; such a data type is known as an abstract data type. A program is itself specified as an abstract data type, by considering it to consist of a set of operations on a class of states which model the program variables. The notion of state is, therefore, made explicit

in VDM; this is in contrast to other specification methods such as those of the algebraic approach.

A number of data types and constructs are considered as primitives in VDM. These are familiar mathematical objects such as sets, lists, mappings, abstract syntax and functions. In addition, the notation of first order logic is used extensively. The following sections describe these notations briefly. Section 5.6 describes the way abstract data types are specified, developed and verified. The chapter ends with an example which illustrates the method in practice.

The actual notation that we shall use below is that of EPROL whose syntax is slightly different to that of VDM, but essentially the same in meaning. Certain constructs and notions (e.g. polymorphic types) are peculiar to EPROL and do not exist in VDM.

5.2 LOGIC

The notation of logic is based on a simple set containing two elements only. These elements are TRUE and FALSE. The set is called Bool, so:

```
Bool = {TRUE, FALSE}
```

TRUE and FALSE are often called *truth values*. Every expression in logic (also called a boolean expression or predicate) has a truth value. Logic provides a number of operators, usually referred to as *boolean operators*, for writing predicates. These are: not, and, or, implication and equivalence operators, represented by the symbols ~, &, |, ==> and <=> respectively. The boolean operators have the following meanings:

~x	is true if x is false, and false otherwise.
x & y	is true if both x and y are true, and false otherwise.
x | y	is false if both x and y are false, and true otherwise.
x ==> y	is false if x is true and y is false, and true otherwise.
x <=> y	is true if both x and y are either true or false, and false otherwise.

It follows, therefore, that:

```
x ==> y ≡ ~x | y
x <=> y ≡ (x & y) | (~x & ~y)
```

quantifiers

Occasionally in logic, we would like to state that a certain predicate holds for various values of some variable. This is where quantifiers may be useful. There are two quantifiers in logic called the *universal* and the *existential* quantifier, represented here by the symbols `.A` and `.E` respectively. Predicates written using quantifiers are called *quantified expressions*; examples are:

```
(.A x £ s: p(x))
(.E x £ s: p(x))
```

where `s` is a set and `p` is a predicate over `x`. The former expression states that for any `x` in `s`, `p(x)` is true. The latter expression states that there is a `x` in `s` such that `p(x)` is true. It follows, therefore, that:

```
~(.A x £ s: p(x)) <=> (.E x £ s: ~p(x))
~(.E x £ s: p(x)) <=> (.A x £ s: ~p(x))
```

A special case of the existential quantifier is the *unique existential* quantifier represented by `.E!`; for example,

```
(.E! x £ s: p(x))
```

states that there is a unique `x` is `s` such that `p(x)` holds.

In the above examples, `x` is called a *bound variable*; `s` is called a *constraint*; and `p` is called the *body* of the quantified expression. In general, a quantified expression may have more than one bound variable. Such expressions can always be written as a sequence of nested quantified expressions with single bound variables; for example:

$$(.A\ x_1, x_2, \ldots, x_n\ £\ S:\ p) \quad <=> \quad (.A\ x_1\ £\ S:\ (.A\ x_2\ £\ S:\ \ldots\ (.A\ x_n\ £\ S:\ p)\ldots))$$

5.3 ABSTRACT OBJECTS

This section briefly describes certain abstract object classes which are used extensively in specifications. Each object class will be described briefly, and informally, together with its associated set of operators.

sets

A set is an unordered collection of objects with no repetitions. An object in a set is said to be a *member* of that set. A set may be defined *explicitly* by enumerating all its members. For example,

```
{Japan, Italy, Canada, Germany}
```

specifies a set of countries. Sets which consist of a range of integers may be abbreviated to a range; for example:

```
{10, 11, 12, 13, 14} = {10:14}
```

A set may also be defined *implicitly*, by defining a general member of the set. For example,

```
{sqrt(x): x £ s & is_even(x)}
```

specifies the set of square roots of even integers in `s`. In general, an implicit set is written as:

```
{e(x1,...,xn): p(x1,...,xn)}
```

where `e` is an expression called the *generator* and `p` is a predicate called the *constraint*, both over variables `x1, ..., xn` which are called the *bound variables.*

The set operators are summarised in figure 5.1 and have the following meanings:

`e £ s`	is true if `e` is a member of `s`, and false otherwise.
`s1 .S. s2`	is true if `s1` is a subset of `s2`, and false otherwise. formally, `s1 .S. s2 <=> (.A e £ s1: e £ s2)`
`s1 .P. s2`	is true if `s1` is a proper subset of `s2`, and false otherwise. formally, `s1 .P. s2 <=> s1 .S. s2 & s1 /= s2`
`s1 .U. s2`	denotes the union of `s1` and `s2` (i.e. the set of objects which are either in `s1`, or in `s2`, or both.) formally, `s1 .U. s2 = {e: e £ s1 | e £ s2}`
`s1 .I. s2`	denotes the intersection of `s1` and `s2` (i.e. the set of objects which are in both `s1` and `s2`.) formally, `s1 .I. s2 = {e: e £ s1 & e £ s2}`
`s1 - s2`	denotes the difference of `s1` and `s2` (i.e. the set of objects which are in `s1` but not in `s2`.) formally, `s1 - s2 = {e: e £ s & ~(e £ s2)}`
card `s`	denotes the cardinality of `s` (i.e. the number of members of `s`.)
power `s`	denotes the power set of `s` (i.e. the set of all subsets of `s`.) formally, *power* `s = {e: e .S. s}`
union `ss`	denotes the distributed union of `ss` (i.e. the union of all sets in a set of sets `ss`.) formally, *union* `ss = {e: (.E s £ ss: e £ s)}`

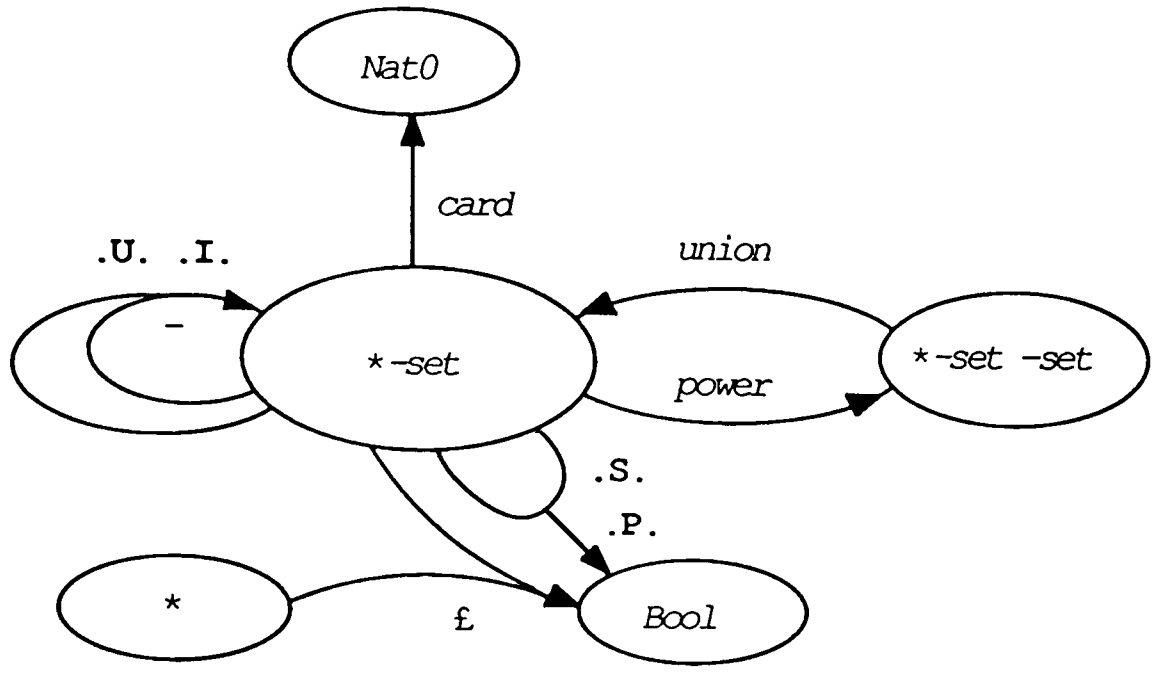


FIGURE 5.1 Summary of set operators.

Two additional operators are selection and unique selection, represented by ! and ! ! respectively. These have a similar syntax to quantifiers; for example,

```
(! x £ s: p(x))
```

produces an element of s (if any) for which p holds, in a pseudo non-deterministic manner. Similarly,

```
(!! x £ s: q(x))
```

produces the unique element of s (if any) for which q holds.

lists

A list is an ordered collection of objects which may contain repetitions. An object in a list is said to be an *element* of that list. Like sets, a list may be defined explicitly by enumerating all its elements. For example,

```
<Austin, Fiat, Rover, Fiat, Ford>
```

specifies a list of cars. Alternatively, a list may be defined implicitly. For example,

```
<i: i £ s & (.A j £ {2:i}: i%j /= 0)>
```

produces the list of all those integers in s which are prime (% is the remainder operator.) In general, an implicit list definition is written as:

```
<e(x1,...,xn): p(x1,...,xn)>
```

and is similar to an implicit set definition.

The list operators are summarised in figure 5.2 and have the following meanings:

`l[i]`	denotes the `i`-th element of list `l` (starting at `1`.)
`l1 || l2`	denotes the concatenation of `l1` and `l2` (i.e. the list consisting of elements of `l1` followed by elements of `l2`, in the same order as `l1` and `l2` and having a length equal to the length of `l1` plus length of `l2`.)
hd `l`	denotes the head of `l` (i.e. the first element of `l`.) formally, *hd* `l = l[1]`
tl `l`	denotes the tail of `l` (i.e. the list consisting of all elements of `l` except the first, in the same order as `l`.) formally, *hd* `l || tl l = l`
len `l`	denotes the length of `l` (i.e. the number of elements of `l` including the repetitions, if any.)
elems `l`	denotes the elements of `l` (i.e. the set consisting of elements of `l`.)
inds `l`	denotes the set of indices of `l`. formally, *inds* `l = {1:len l}`
conc `ll`	denotes the distributed concatenation of the lists in the list of lists `ll`. formally, `ll = <l1,l2,...,ln> <=>` *conc* `ll = l1||l2||...||ln`

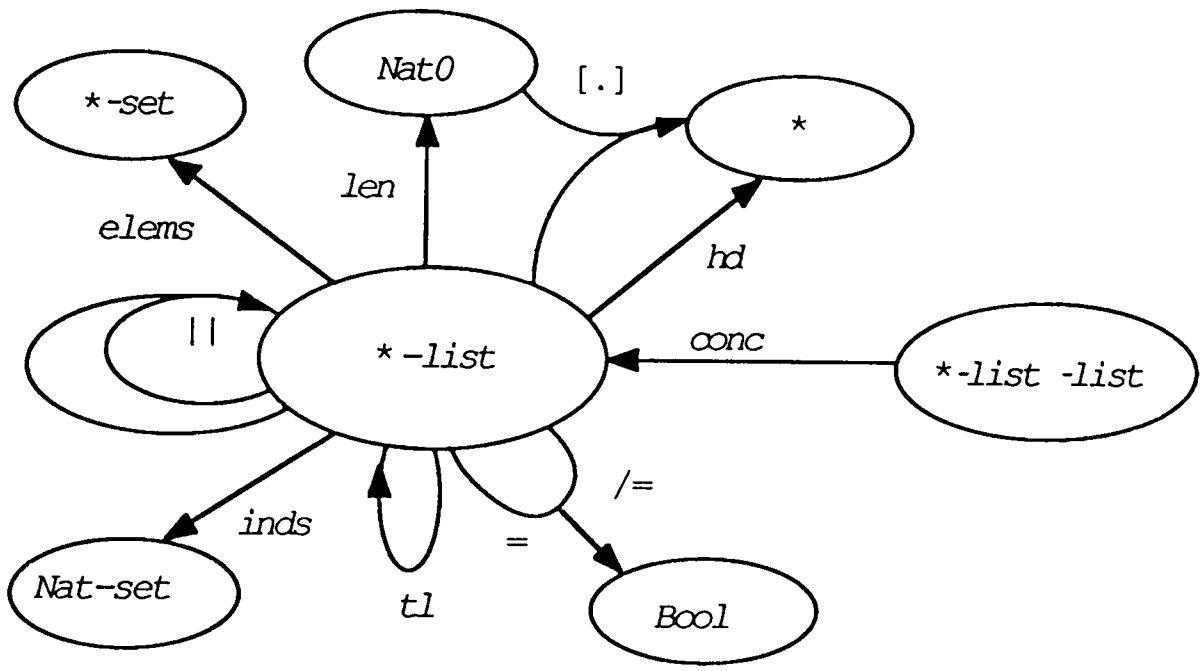


FIGURE 5.2 Summary of list operators.

Two additional list operators are `map` and `dist`. These have an unusual syntax and are used for mapping or distributing an operator (or a function) over a list, where the operator (or function) must be unary or binary for `map`, and binary for `dist`. Here are two examples of their use:

```
map(card: <{},{1,2},{5}>) = <0,2,1>
dist(+: <5,10,20>) = 35
```

Combination of `map` and `dist` provides a succinct notation for specification. For example, a predicate denoting that the elements of a list of numbers are sorted in ascending order may be written as:

```
dist(&,<: l)
```

which is equivalent to:

```
dist(&: map(<: l))
```

Nested `map` and `dist` applications may be abbreviated according to the following conventions:

```
    map(f1,f2,...,fn: l)
  = map(f1: map(f2: ... map(fn: l)...))
```
and
```
    dist(f,g1,...,gn: l)
  = dist(f: map(g1: ... map(gn: l)...))
```

where `f`'s and `g`'s may be operators or functions.

mappings

A mapping (or map) is a finite function. It maps the elements of a set, called its *domain*, to the elements of a set, called its *range*. A mapping can be defined explicitly by enumerating how individual elements of its domain are mapped into individual elements of its range. For example,

```
[John -> 20, Peter -> 12, Steve -> 25]
```

maps three persons to their ages. As with sets and lists, a mapping can also be defined implicitly. For example,

```
[i -> i**2: i £ s]
```

maps every number in `s` to its square. In general, an implicit mapping is written as:

```
[e1(x1,...xn) -> e2(x1,...,xn): p(x1,...,xn)]
```

The mapping operators are summarised in figure 5.3 and have the following meanings:

`m(x)`	denotes an element of the range of `m` to which `x` is mapped by `m`.
`m1 + m2`	denotes the mapping which is the result of merging `m1` and `m2` provided the domains of `m1` and `m2` are disjoint. formally, `m1 + m2 = [e->f: e £ dom m1 & f = m1(e) | e £ dom m2 & f = m2(e)]`
`m1 ++ m2`	denotes the mapping which is the result of overwriting `m1` by `m2`. formally, `m1 ++ m2 = [e->f: e £ (dom m1-dom m2) & f = m1(e) | e £ dom m2 & f = m2(e)]`
`m1 ^ m2`	denotes the composition of `m1` and `m2` provided the range of `m2` is a subset of the domain of `m1`. formally, `m1 ^ m2 = [e->f: e £ dom m2 & f = m1(m2(e))]`
`m /+ s`	denotes the mapping which is identical to `m` but whose domain is restricted to the set `s`.

formally, `m /+ s = [e->m(x): x £ (dom m .I. s)]`

`m /- s` denotes the mapping which is identical to `m` but from whose domain the elements of the set `s` have been removed.
formally, `m /- s = [e->m(x): x £ (dom m - s)]`

dom `m` denotes the domain of `m`.

rng `m` denotes the range of `m`.

merge `ms` denotes the distributed merge of the mappings in the set of mappings `ms`, provided the domains of the mappings are disjoint.
formally, `merge ms = [e->(!! m £ ms: e £ dom m)(e)`
`: e £ union {dom m: m £ ms}]`

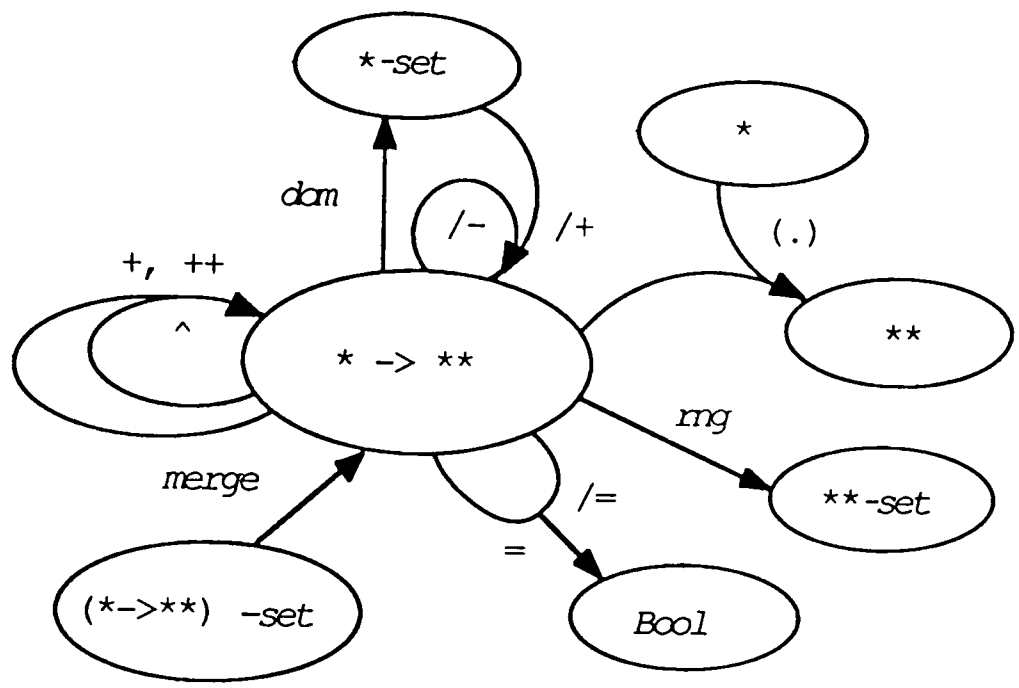


FIGURE 5.3 Summary of mapping operators.

5.4 ABSTRACT SYNTAX

Certain elementary domains are predefined in EPROL; these are:

Nat	- Natural numbers.
Nat0	- Natural numbers including zero.
Int	- Integer numbers.
Real	- Real numbers.
Bool	- Booleans, i.e. {*TRUE*, *FALSE*}.
Char	- Characters.
Str	- Strings.

The notation of abstract syntax allows one to define other, possibly more complex domains. An abstract syntax definition consists of one or more abstract syntax rules. A rule has the general form:

```
domain_id = domain_expr
```

which introduces a new domain called `domain_id`, denoted by `domain_expr`. A domain expression is an expression consisting of domain names and domain operators. The

operators of the abstract syntax notation are as follows:

`D-set`	denotes a class of objects where each object is a subset of `D`. formally, `s £ D-set <=> s .S. D`
`D-list`	denotes a class of objects where each object is a list of some objects in `D`. formally, `l £ D-list <=> elems l .S. D`
`D1 -> D2`	denotes a class of objects where each object is a mapping whose domain is a subset of `D1` and whose range is a subset of `D2`. formally, `m £ (D1 -> D2) <=> dom m .S. D1 & rng m .S. D2`
`D1 | D2`	denotes a class of objects where each object is either in `D1`, or in `D2`, or both. formally, `d £ (D1 | D2) <=> d £ D | d £ D2`
`[D]`	denotes a class of objects where each object is either in `D` or is just the `NIL` object. formally, `d £ [D] <=> d £ D | d = NIL`

Round brackets may also be used in domain expressions for grouping and to enhance readability. Two domain definition examples are given below:

```
D1 = Int-set -> Bool-list-set
D2 = (D1 -> Str) -> (D1 | Int | Real)
```

Each object in `D1` is a mapping from the power set of integers to the sets of lists of booleans. Each object in `D2` is a mapping from the mappings which map `D1` to strings, to either an object in `D1`, or an integer, or a real.

Abstract syntax rules may also be recursive (i.e. refer to themselves.) For example,

```
D = Int -> [D]
```

defines a domain called `D`, where each object in `D` is a mapping from integers to either `D` itself, or to `NIL`.

trees

The notation so far described does not allow us to define structured objects. An structured object is an object which consists of a number of components. The domain of such objects is called a *tree*. These are specified by replacing the = symbol, in an abstract syntax definition, by the symbol ::. For example,

```
D :: Int, Str-list, Real-set
```

defines a tree domain called `D` where each object in `D` has exactly three components. These being, in order, an integer, a string list, and a real set. An object in a tree domain is usually called a tree *branch*. A special function called `mk` may be used to make such objects, e.g.:

```
mk-D(1,<"ab","ef">,{1.5}) £ D
```

Individual components of a tree domain may be named as shown below:

```
D :: .i: Int, .sl: Str-list, .rs: Real-set
```

Given this domain definition, individual components of an object in D may be specified by adding the component name to the end of an object. For example, let

```
d = mk-D(2,<"hi","there">,{1.5})
```

then

```
d.i = 2
d.sl = <"hi","there">
d.rs = {1.5}
```

It is also possible to name only selected components:

```
D :: Int, .sl: Str-list, Real-set
```

A tree may also be defined using the following notation

```
D = tree Int, Str-list, Real-set
```

which is equivalent to

```
D :: Int, Str-list, Real-set
```

The former form is useful for defining nested trees, sets of trees etc., in a single abstract syntax rule. For instance,

```
P = tree .n: Str, .a: Int, .p: (tree str-set, Int)
```

is a shorter way of saying:

```
P :: .n: Str, .a: Int, .p: Q
Q :: Str-set, Int
```

and avoiding the definition of a new domain Q.

5.5 COMBINATORS

EPROL provides a number of combinators for use in specifications. These do not cause any side-effects; they simply return a value. Each combinator is informally and briefly described below.

the let expression

The let expression is used for naming one or more expressions within another expression. The simplest form of a let expression is:

```
let id = expr1 in
    expr2
```

which means that every occurrence of `id` in $expr_2$ will be bound to the value of $expr_1$. More generally,

```
let id1 = expr1,
    id2 = expr2,
     :
    idn = exprn in
    expr
```

binds id_1, id_2, ..., id_n to $expr_1$, $expr_2$, ..., $expr_n$ respectively, and in parallel, in `expr`.

The let combinator may also be used for naming individual fields of a tree. Consider the following abstract syntax definition:

```
Student :: .name: Str, .age: Nat, .id: Nat0;
```

Now suppose `st £ Student` and that `st = ("Phil", 25, 10516)` then:

```
let (n,a,id) = st in
    n = "Phil" & a = 25 & id = 10516
```

is true. In this example `n` is bound to the first field in the tree (i.e. `"Phil"`), `a` to the second field (i.e. `25`) and `id` to the last field (i.e. `10516`).

the if-then-else expression

The simplest form of a conditional expression is the if-then-else expression. The general form for this combinator is:

```
if bool_expr then expr1
else expr2
```

The overall value of this expression is the value of `expr1` if `bool_expr` evaluates to TRUE, and the value of `expr2` if it evaluates to FALSE.

the mac expression

This is McCarthy's expression and has the general form:

```
mac {
  bool_expr1 => expr1,
  bool_expr2 => expr2,
   :
  bool_exprn => exprn,
}
```

which provides a multi-branch conditional expression. The `bool_exprs` on the left hand side are evaluated in the order they appear. If $\texttt{bool\_expr}_i$ evaluates to TRUE then $\texttt{expr}_i$ will be evaluated and its value will be returned as the value of the overall mac expression. At least one $\texttt{bool\_expr}_i$ must evaluate to TRUE.

the cases expression

This is similar to the mac expression and has the general form:

```
cases exprs {
   lexpr1 => rexpr1,
   lexpr2 => rexpr2
    :
   lexprn => rexprn,
}
```

First $\texttt{expr}_s$ is evaluated. Then `lexpr`'s are evaluated in the order they occur. If $\texttt{lexpr}_i = \texttt{expr}_s$ then $\texttt{rexpr}_i$ will be evaluated and its value will be returned as the value of the cases expression. At least one $\texttt{lexpr}_i$ must have the same value as $\texttt{expr}_s$. As a convention, the last `lexpr` may be simply TRUE to ensure this.

5.6 ABSTRACT DATA TYPES

To be concise when specifying software systems, one must depart from the elementary data types of a specification language and instead 'create' data types which match the problem at hand more closely and more naturally. Such a data type is known as an *abstract data type* and is characterised by its private set of operations.

specification

An abstract data type is specified by a class of states and a set of private operations to act upon the states. It introduces a new data type, where an object of this type can be manipulated only through the specified operations. The class of states is denoted by a domain usually restricted by a predicate, called the data type invariant, which must be preserved by the operations.

In EPROL, abstract data types are specified by ADT modules. The general structure of an ADT module is shown in figure 5.4.

```
ADT adt_id
  DOM  ... private domain definitions ...
  TYPE ... private type clause definitions ...
  AUX  ... private auxiliary function definitions ...
  OPS
    :
    private operation definitions
    :
END adt_id
```

FIGURE 5.4 The general structure of an ADT module.

The first three parts in the definition (i.e. DOM, TYPE and AUX) are optional. The DOM part introduces new domains; for example,

```
DOM Product = Pname -> Pid;
    Pname = Str;
    Pid = Int;
```

defines three new domains called `Product`, `Pname` and `Pid`. The object class (i.e. class of states) for the abstract data type itself must be defined here.

The AUX part is used for defining new auxiliary functions. Every function defined here must have its type clause already defined in the TYPE part. For example,

```
TYPE is_disjoint: Int-set, Int-set --> Bool;
     is_empty: Int-set --> Bool;
AUX  is_disjoint(s1,s2) == s1 .I. s2 = {};
     is_empty(s) == s = {};
```

defines two auxiliary functions called `is_disjoint` and `is_empty`. The domain of a function can be restricted by a pre-condition; this is a predicate over the domain of the

function which must hold before the function is applied. For example, a function called max which takes a set of integers and returns the largest integer in the set may be defined as:

```
TYPE max: Int-set --> Int;
AUX  pre-max(s) == s /= {};
     max(s) == (! e £ s: (.A e1 £ s: e >= e1));
```

where the pre-condition indicates that max is not defined for the empty set.

If an abstract data type has a data type invariant then it must be defined in the AUX part. For example,

```
AUX inv-Product(p) == dom p /= {};
```

defines a data type invariant for an abstract data type called Product.

The last part in an ADT module is OPS and specifies one or more operations for the abstract data type. The general structure of an operation specification is shown in figure 5.5. Each operation acts upon the class of states of the abstract data type. In addition, an operation may take arguments and/or return a result, as in a function. This is specified by the type clause of the operation. For example,

```
OP: Dom1, Dom2 --> Dom3
```

specifies that operation OP takes two arguments in Dom1 and Dom2 and returns a result in Dom3.

```
OP_ID: ... operation type clause ...;
  pre(...) == ... pre-condition predicate ...;
  post(...) == ... post-condition predicate ...;

    :
    private auxiliary function definitions
    :
END OP_ID
```

FIGURE 5.5 The general structure of an operation specification.

An operation is specified in terms of two predicates called the pre and the post-condition. The pre-condition is optional and is assumed to be TRUE if it is not present. It is a predicate over the states and any arguments the operation may take, and specifies a condition which must hold before the operation is applied. Alternatively, it may be specified as a list of exception clauses, where each exception clause maps a predicate to an exception

name. The overall pre-condition will then be a conjunction of the negation of individual exception predicates (see appendix D.1.)

The post-condition specifies a condition which must hold after the operation is applied. It is a predicate over the states before and after the operation is applied, and any arguments and result the operation may take and produce. Consider a general operation OP over the states St with the following type clause:

```
OP: Dom1, Dom2, ..., Domn --> Res;
```

The pre and post-conditions will have the following implicit type clauses:

```
pre:  St, Dom1, Dom2, ..., Domn --> Bool;
post: St, Dom1, Dom2, ..., Domn, St, Res --> Bool;
```

The position of a parameter in a condition indicates its actual domain. So, for example,

```
pre(st,arg1,arg2,...,argn) == ... ;
post(st,arg1,arg2,...,argn,st',res) == ... ;
```

indicates that

```
st, st' £ St
arg1 £ Dom1, arg2 £ Dom2, ..., argn £ Domn
res £ Res
```

where st and st' refer to the 'states before' and 'states after' the operation is applied respectively.

In EPROL, a parameter can be replaced by a minus symbol according to the following conventions:

- When replacing the 'states before' parameter it implies that we are not interested in the value of the states before the operation is applied.
- When replacing the 'states after' parameter it implies that the value of the states will not be changed by the operation.
- When replacing any other parameter it implies that the value of that parameter is not relevant to the specified condition.

An operation can also have its own private auxiliary functions. Such functions may appear inside an operation specification, just after the post-condition.

refinement

The initial specification of an abstract data type is written as abstractly as possible while ensuring that it captures all the required properties of the problem at hand. The abstract data type is then developed by the process of *data refinement*, whereby it is realised in a more concrete form. This process produces a so called *representation* of the abstract data type. The process consists of four steps:

- Find a more concrete class of states for the abstract data type.
- Redefine the data type invariant for the new class of states.
- Find a function which maps each object in the new class of states to its corresponding object in the previous class of states. This is called a *retrieve* function and relates a representation to its abstraction.
- Redefine each operation of the abstract data type for the new class of states.

Refinement is an iterative process which results in successively more concrete representations of an abstract data type. It is repeated until the final representation is in a sufficiently concrete form.

The original specification and each subsequent refinement can be shown to be internally correct. In addition, one can show that a representation is correct with respect to its abstraction.

verification rules

VDM provides a number of useful rules for verifying the correctness of abstract data type specifications and their refinements. These are directly used in EPROL and are briefly described below. For a more detailed discussion of the rules see [Jones80] or [Jones86].

Let `D` be an abstract data type, having a class of states `S`, a data type invariant `inv` and a set of operations `P1`, `P2`, ..., `Pn`. Operation `Pi` is *valid* if it preserves the data type invariant, i.e. for any `s` in `S`:

```
pre-Pi(s,args) & inv(s) & post-Pi(s,args,s',res) ==> inv(s')     [R1]
```

Let `D1` be a refinement of `D`, having a class of states `S1`, a data type invariant `inv1` and a set of operations `Q1`, `Q2`, ..., `Qn` corresponding to `P1`, `P2`, ..., `Pn` respectively. Also, let

retr be the retrieve function from S1 to S:

```
retr: S1 --> S
```

The retrieve function is *total* over valid states (i.e. states which satisfy the data type invariant inv1) if:

```
(.A s1 £ S1: inv1(s1) ==> (.E s £ S: s = retr(s1)) & inv(retr(s1)))
                                                                    [R2]
```

and S1 is an *adequate* representation of S if:

```
(.A s £ S: inv(s) ==> (.E s1 £ S1: inv1(s1) & s = retr(s1)))    [R3]
```

Operation Qi *models* operation Pi if:

```
(.A s1 £ S1: inv1(s1) & pre-Pi(retr(s1),args) ==> pre-Qi(s1,args))
                                                                    [R4]
```

and

```
(.A s1 £ S1: inv1(s1) & pre-Qi(s1,args) & post(s1,args,s1',res)
             ==> post-Pi(retr(s1),args,retr(s1'),res))          [R5]
```

Proof of correctness reduces to showing that R1 holds for each specification and that R2-R5 hold for each specification with respect to its abstraction.

polymorphic types

All abstractions so far described such as functions, operations and abstract data types required a precise domain specification. This in turn limits their utility. Consider, for example, a function called largest which returns the largest set in a set of sets and has the following definition:

```
largest: Int-set-set --> Int-set;
pre-largest(ss) == ss /= {};
largest(ss) == (! s £ ss: (.A s1 £ ss: card s >= card s1));
```

This function is only valid for sets of integer sets.

More desirable, and certainly more useful, would be a function which would work for sets of sets of any type. Such functions may be defined using polymorphic types [Turner85]. The function largest, for example, may be defined polymorphicly by defining its type clause to be:

```
largest: *-set-set --> *-set;
```

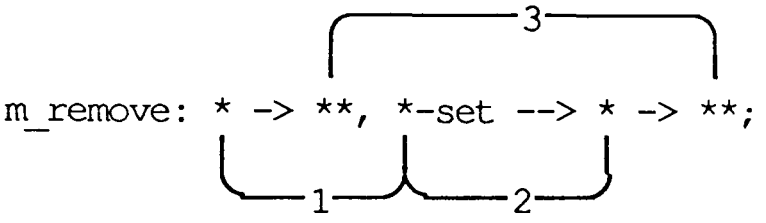
More complicated functions may require more than one polymorphic type. Additional types may be specified by **, *** and so on. To give an example, suppose we were to define the map remove operator /- as a function. It may be defined as the following polymorphic function:

```
m_remove: * -> **, *-set --> * -> **;
m_remove(m,s) == [x -> m(x) : x £ (dom m - s)]
```

Here ** specifies a type which may be different from the one specified by *, as shown by the following applications of m_remove:

```
  m_remove([1->"small", 10->"big", 100->"very big"], {10})
= [1->"small",100->"very big"]

  m_remove([1->{1}, 2->{1,2}, 3->{1,2,3}], {1,2,5})
= [3->{1,2,3}]
```

In the first example * becomes *Int* and ** becomes *Str*. In the second example * becomes *Int* and ** becomes *Int-set*. Every application of m_remove involves three type checkings, as depicted by the diagram below:



The use of polymorphic types is not restricted to functions. They may also be used in operations, abstract data types and other forms of abstraction described in the rest of this thesis. Polymorphic types are especially suitable for writing general purpose abstractions, which crop up in a variety of specifications, without losing the advantages of type checking.

5.7 A DEVELOPMENT EXAMPLE

This section illustrates, by means of a realistic example, how a formal specification may be developed, refined, evaluated and formally verified in EPROS. The interactive evaluation of a formal specification can serve two purposes. Firstly, it can provide a means of observing the behaviour of the specified system in order to see whether it is indeed the one desired. Secondly, it may serve as a cheap and quick way of detecting design errors. The

example given here exhibits the potential of the approach for both applications.

problem specification

The program to be specified is a software tool that records the relationships between the modules of a software system. An informal statement of the requirements is given below:

> A program is required which records the *uses* and *used-by* relationships between the modules of a software system. It should allow the user to do the following:
>
> - Add a module to the system.
> - Delete a module from the system.
> - List what modules a given module may use.
> - List what modules may use a given module.
> - List all recursive modules.

Let us call this program the 'Cross Usage' program. Our first task is to find a suitable object class that can model the problem. Let us call the object class `Xusage`; it can be modelled by a mapping which maps every module in a system to the set of modules it may use, i.e.:

```
Xusage = Module -> Module-set;
```

At this point, we shall not specify the domain `Module`. Every object in `Module` is understood to correspond to a module in a system. `Module`, in other words, is the set of all possible modules in software systems. As an example, consider the following object in `Xusage`:

```
[mod1 -> {mod2,mod3},
 mod2 -> {},
 mod3 -> {}]
```

It represents a system which has the structure diagram shown in figure 5.6,

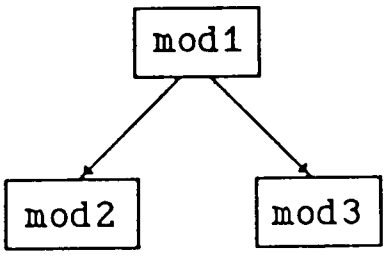


FIGURE 5.6 A simple structure diagram.

where mod1 may call mod2 and mod3, and mod2 and mod3 may not call any other module.

Obviously, given that `s` is the set of modules a module `m` may call then `s` must be

contained in the domain of the mapping, i.e.:

```
(.A m £ dom xu: xu(m) .S. dom xu)
```

where xu is the mapping. This is specified by the following data type invariant:

```
inv-Xusage(xu) == union rng xu .S. dom xu
```

Now let us specify the operations of abstract data type Xusage. The first operation is very simple; it just initialises the mapping to an empty map:

```
INIT: --> ;
   post(-,xu') == xu' = [];
END INIT
```

The next operation to be specified, adds a module to the system. It will take a module and a set of modules that the module may use and adds them to the mapping:

```
ADD_MOD: Module, Module-set --> ;
   pre(xu,mod,-) == ~(mod £ dom xu) | xu(mod) = {};
   post(xu,mod,uses,xu') ==
        xu' = xu +  [m -> {}: m £ (uses - dom xu)]
                 ++ [mod -> uses];
END ADD_MOD
```

The pre-condition specifies that either the module (to be added) should not be already in the mapping or, if it is, it should not be using any other module at the moment. The post-condition specifies that the mapping after the operation is applied will be equal to the mapping before the operation is applied, merged with an implicit mapping which maps each new module in uses to the empty set, and then overwritten by an explicit mapping which maps the module to be added to uses.

The next operation deletes a module from the system:

```
DEL_MOD: Module --> ;
   pre(xu,mod) == mod £ dom xu;
   post(xu,mod,xu') ==
        xu' = [m -> xu(m) - {mod} : m £ (dom xu - {mod})];
END DEL_MOD
```

Obviously a module to be deleted must already be present in the system, hence the pre-condition. The post-condition specifies that the effect of the operation will be that all occurrences of the deleted module will be removed from the right hand side of the mapping and the particular entry for the module itself will be totally removed from the mapping.

The next two operations are trivial. The first is USES and returns the set of modules given module may use:

```
USES: Module --> Module-set;
   pre(xu,mod) == mod £ dom xu;
   post(xu,mod,-,ms) == ms = xu(mod);
END USES
```

The second operation, USED_BY, returns the set of modules that may use a given module:

```
USED_BY: Module --> Module-set;
   pre(xu,mod) == mod £ dom xu;
   post(xu,mod,-,ms) ==
       ms = {m : m £ dom xu & mod £ xu(m)};
END USED_BY
```

The last operation produces the set of recursive modules in a system:

```
REC_MOD: --> Module-set;
   post(xu,-,ms) ==
       ms = {m: m £ dom xu & reaches(m,m,xu)};

   pre-reaches(m1,m2,xu) == m1 £ dom xu & m2 £ dom xu;
   reaches(m1,m2,xu) ==
         m2 £ xu(m1) | (.E m £ xu(m1): reaches(m,m2,xu));
END REC_MOD
```

The post-condition uses an auxiliary function called `reaches` which has the following typ clause:

```
reaches: Module, Module, Xusage --> Bool;
```

This function is defined to be local to the operation and returns TRUE if a module can reacl another module through a sequence of one or more calls. We observe the conciseness of th post-condition: the set of modules m in the system such that m can somehow reach itself.

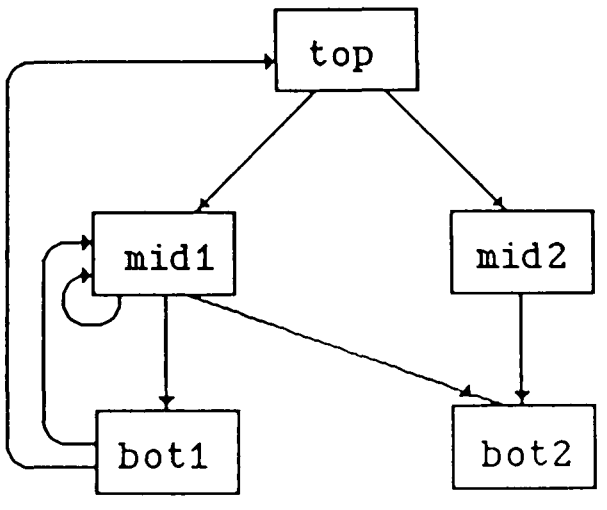


FIGURE 5.7 A simple software system.

Having specified our abstract data type the next stage in the development process involves the evaluation of the specification. Here, we examine the behaviour of the system by executing the specification. The following is a simple evaluation session based on setting up and manipulating the system of figure 5.7. The domain `Module` is assumed to be *Str*. EPROS response is printed in bold.

```
epi> VAR x: Xusage;;                    /* define x to be of type Xusage */
epi> INIT(x);                           /* initialise x */
epi> x;                                 /* check the contents of x */
[]
epi> ADD_MOD(x,"top",{"mid1","mid2"});          /* add modules */
epi> ADD_MOD(x,"mid1",{"mid1","bot1","bot2"});
epi> ADD_MOD(x,"mid2",{"bot2"});
epi> ADD_MOD(x,"bot1",{"top","mid1"});
epi> x;                                 /* check the contents of x */
["bot2" -> {},
 "mid2" -> {"bot2"},
 "top"  -> {"mid2","mid1"},
 "mid1" -> {"bot2","bot1","mid1"},
 "bot1" -> {"mid1","top"}]
epi> USES(x,"top");                     /* list modules that top uses */
{"mid1","mid2"}
epi> USED_BY(x,"mid1");                 /* list modules that use mid1 */
{"top","mid1","bot1"}
epi> REC_MOD(x);                        /* list recursive modules */
{"top","mid1","bot1"}
epi> DEL_MOD(x,"mid2");                 /* delete module mid2 */
epi> x;                                 /* check the contents of x */
["bot2" -> {},
 "top"  -> {"mid1"},
 "mid1" -> {"bot2","bot1","mid1"},
 "bot1" -> {"mid1","top"}]
```

Having convinced ourselves that the exhibited behaviour is indeed the one desired, we then verify the specification. Since this is the first specification of the system, all that we can verify at this stage is validity, i.e. that each operation preserves the data type invariant.

Theorem 5.1: Operation `INIT` is valid.

Proof: For this operation we observe that:

```
post(xu,xu') ==> inv(xu')                                        ---- (1)
```

since

```
inv(xu') = inv([])
         = union rng [] .S. dom []
         = {} .S. {} = TRUE
```

From (1) it follows that:

```
pre(xu) & inv(xu) & post(xu,xu') ==> inv(xu')
```

which proves that INIT is valid.

Theorem 5.2: Operation ADD_MOD is valid.

Proof: We must show that:

```
pre(xu,mod,uses) & inv(xu) & post(xu,mod,uses,xu') => inv(xu')
```

Now

```
inv(xu') = inv(xu + m1 ++ m2)
```

where

```
m1 = [m -> {}: m £ (uses - dom xu)]
and m2 = [mod -> uses]
```

Hence

```
inv(xu') = inv(xu + m1 ++ m2)
= union rng (xu + m1 ++ m2) .S. dom (xu + m1 ++ m2)
```

From *pre* it follows that

```
union rng (xu + m1 ++ m2) = union rng xu .U. union rng m1 ++ m2
```

Hence

```
inv(xu') =
= (union rng xu .U. union rng (m1 ++ m2)) .S. dom (xu + m1 ++ m2)
= (union rng xu .U. uses).S.(dom xu .U. dom m1 .U. dom m2)
= (union rng xu .U. uses).S.(dom xu .U. (uses-dom xu) .U. {mod})
= (union rng xu .U. uses).S.(dom xu .U. uses .U. {mod})
```

which is true since using *inv*(xu):

```
union rng xu .S. dom xu
```

and completes the proof.

Theorem 5.3: Operation DEL_MOD is valid.

Proof: Consider the stronger condition:

```
inv(xu) & post(xu,mod,xu') ==> inv(xu')
```

where

```
inv(xu') == union rng xu' .S. dom xu'
```

Now (using *post*):

```
union rng xu' .S. union rng xu - {mod}                    ---- (1)
```

and (using *inv*):

```
union rng xu - {mod} .S. dom xu - {mod}                   ---- (2)
```

also (using *post*):

```
dom xu - {mod} = dom xu'                                  ---- (3)
```

From (1),(2) and (3) it follows that:

```
union rng xu' .S. dom xu'
```

hence

```
inv(xu') = TRUE
```

which completes the proof.

The last three operations require no proof since they do not change the states and, therefore, are always valid. The complete specification of abstract data type Xusage is shown in figure 5.8.

```
ADT Xusage
  DOM Xusage = Module -> Module-set;
  TYPE reaches: Module, Module, Xusage --> Bool;
  AUX inv-Xusage(xu) == union rng xu .S. dom xu;
  OPS
    INIT: -->;
       post(-,xu') == xu' = [];
    END INIT

    ADD_MOD: Module, Module-set -->;
       pre(xu,mod,-) == ~(mod £ dom xu) | xu(mod) = {};
       post(xu,mod,uses,xu') ==
            xu' = xu +  [m -> {} : m £ (uses - dom xu)]
                     ++ [mod -> uses];
    END ADD_MOD

    DEL_MOD: Module -->;
       pre(xu,mod) == mod £ dom xu;
       post(xu,mod,xu') ==
            xu' = [m -> xu(m)-{mod} : m £ (dom xu - {mod})];
    END DEL_MOD

    USES: Module --> Module-set;
       pre(xu,mod) == mod £ dom xu;
       post(xu,mod,-,ms) == ms = xu(mod);
    END USES

    USED_BY: Module --> Module-set;
       pre(xu,mod) == mod £ dom xu;
       post(xu,mod,-,ms) ==
            ms = {m : m £ dom xu & mod £ xu(m)};
    END USED_BY

    REC_MOD: --> Module-set;
       post(xu,-,ms) ==
            ms = {m : m £ dom xu & reaches(m,m,xu)};

       pre-reaches(m1,m2,xu) == m1 £ dom xu & m2 £ dom xu;
       reaches(m1,m2,xu) == m2 £ xu(m1) |
                            (.E m £ xu(m1): reaches(m,m2,xu));
    END REC_MOD
END Xusage
```

FIGURE 5.8 Specification of abstract data type Xusage.

refinement of the specification

Having completed the first specification of the system, the next stage involves refining the specification. First we must choose a new, more concrete, object class for the abstract data type. A number of possibilities exist; we suggest the following:

```
Xusage1 = Module -> Cross;
Cross :: .u: Module-set, .b: Module-set;
```

`Xusage1` is the domain of mappings from `Module` to a new domain called `Cross`. Every object in `Cross` has two components. The first component denotes the set of modules a module may use; the second component denotes the set of modules which may use that module. So, for example, the structure diagram in figure 5.6 will be represented by the following mapping in `Xusage1`:

```
[mod1 -> ({mod2,mod3},{}),
 mod2 -> ({},{mod1}),
 mod3 -> ({},{mod1})]
```

What we have done in fact is that we have introduced some redundancy in our model by explicitly including, for each module, the set of modules which may use that module. This is a *design decision*. The refinement process typically involves making one or more design decisions at each stage.

Every design decision must have some justification. The design decision above was made with the hope of gaining some conceptual efficiency in the system. We observe that the introduced redundancy may simplify some operations (e.g. USED_BY) at the cost of making other operations more complicated (e.g. ADD_MOD). In systems that deal with information storage and retrieval usually one requires the retrieve operations to be considerably simpler than the storage operations for the simple reason that the former are used much more often than the latter. This is the basis of the design decision we have made here.

Our next task is to strengthen the data type invariant to preserve the meaning of the problem. The new data type invariant is:

```
inv-Xusage1(xu) ==
    (.A m £ dom xu: (.A m1 £ xu(m).u: m £ xu(m1).b) &
                    (.A m1 £ xu(m).b: m £ xu(m1).u) );
```

This simply states that the following must hold for every module m in the system: if m uses a module m1 then m should be in the set of modules that use m1 in the mapping, and that if m is used by a module m1 then m should be in the set of modules that m1 uses in the mapping. This is shown diagrammatically in figure 5.9 for the system in figure 5.6.

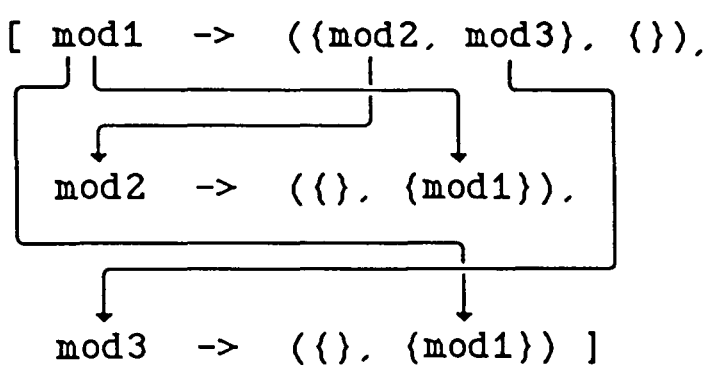


FIGURE 5.9 A Diagrammatic view of *inv*-Xusage1.

The relationship between Xusage and Xusage1 is documented by the following retrieve function:

```
retr: Xusage1 --> Xusage;
retr(xu1) == [m -> xu1(m).u: m £ dom xu1];
```

We now show that retr is total over valid states and that Xusage1 is an adequate representation of Xusage.

Theorem 5.4:
```
(.A x1 £ Xusage1: inv-Xusage1(x1) ==>
      (.E x £ Xusage: x = retr(x1) &                    ---- (1)
                      inv-Xusage(retr(x1))))            ---- (2)
```

Proof: (1) is immediate from the definition of retr. Consider (2), which reduces to showing that:

```
union rng x .S. dom x
```
where
```
x = [m -> x1(m).u: m £ dom x1]                          ---- (3)
```

Using *inv*-Xusage1:

```
union {x1(m).u: m £ dom x1} .S. dom x1
```

and, using (3) it reduces to:

```
union rng x .S. dom x1
```

which completes the proof since (using *inv*-Xusage1):

```
dom x1 = dom x
```

Theorem 5.5: Xusage1 is an adequate representation of Xusage.

Proof: Let x £ Xusage and *inv-Xusage*(x) = *TRUE* then

```
x1 = [m -> mk-Cross(x(m), {n: n £ dom x & m £ x(n)})
          : m £ dom x]                                    ---- (1)
```

represents x. To prove this, let m £ dom x1, we must show that

```
(.A m1 £ x1(m).u: m £ x1(m1).b)                           ---- (2)
```

and that

```
(.A m1 £ x1(m).b: m £ x1(m1).u)                           ---- (3)
```

Using (1), (2) reduces to:

```
m1 £ x(m) ==> m £ {n: n £ dom x & m1 £ x(n)}
```

which is immediate by considering the case when n = m. Using (1) again, (3) reduces to:

```
m1 £ {n: n £ dom x & m £ x(n)} ==> m £ x(m1)
```

which is immediate by considering the case when n = m1, and completes the proof.

The next step in the refinement process involves producing operations in the representation which model the operations in the original specification. The first operation, INIT, will remain as before. Operation ADD_MOD is modeled by ADD_MOD1:

```
ADD_MOD1: Module, Module-set --> ;
   pre(xu,mod,uses) == ~(mod £ dom xu) | xu(mod).u = {};
   post(xu,mod,uses,xu') == xu' =
       xu ++ [m -> if m £ dom xu then
                      mk-Cross(xu(m).u,xu(m).b .U. {mod})
                   else mk-Cross({},{mod})
                   : m £ uses]
          ++ [mod -> mk-Cross(uses,if mod £ dom xu then
                                      xu(mod).b
                                   else {})];
END ADD_MOD1
```

The pre-condition of ADD_MOD1 is more or less identical to that of ADD_MOD. The post-condition, however, has changed considerably. The explicit mapping in *post* produces the entry for mod itself. The implicit mapping produces an entry for each module in uses. It ensures that for each mapping m in uses, mod is included in the set xu'(m).b.

Let us now evaluate operation ADD_MOD1:

```
epi> VAR x: Xusage1;;
```

```
epi> INIT1(x);
epi> x;
[]
epi> ADD_MOD1(x,"top",{"mid1","mid2"});
epi> ADD_MOD1(x,"mid1",{"mid1","bot1","bot2"});
*** post-state of ADD_MOD1 does not satisfy the invariant
epi> x;
["mid1" -> ({},{"top"}),
 "mid2" -> ({},{"top"}),
 "top" -> ({"mid2","mid1"},{})]
```

This simple evaluation shows that ADD_MOD1 does not preserve the data type invariant. Operation ADD_MOD1 is therefore not valid. If we examine the post-condition of this operation carefully we see that it does not behave properly when mod is recursive; if mod £ *dom* xu then

```
xu'(mod) = mk-Cross(uses, xu(m).b)
```

and if ~(mod £ *dom* xu) then

```
xu'(mod) = mk-Cross(uses,{})
```

Both cases produce wrong results since the second set will not contain mod. This problem is avoided by the following post-condition for ADD_MOD1:

```
post(xu,mod,uses,xu') == xu' =
    xu ++ [m -> if m £ dom xu then
                   mk-Cross(xu(m).u,xu(m).b .U. {mod})
                else mk-Cross({},{mod})
                : m £ uses]
       ++ [mod -> mk-Cross(uses,(if mod £ dom xu then
                                    xu(mod).b
                                 else {})    .U.
                                (if mod £ uses then {mod}
                                 else {}))];
```

Let us now evaluate the new version of ADD_MOD1:

```
epi> VAR x: Xusage1;;
epi> INIT(x);
epi> x;
[]
epi> ADD_MOD(x,"top",{"mid1","mid2"});
epi> ADD_MOD(x,"mid1",{"mid1","bot1","bot2"});
epi> ADD_MOD(x,"mid2",{"bot2"});
epi> ADD_MOD(x,"bot1",{"top","mid1"});
epi> x;
["bot2" -> ({},{"mid1","mid2"}),
 "mid2" -> ({"bot2"},{"top"}),
 "top"  -> ({"mid2","mid1"},{"bot1"}),
 "mid1" -> ({"bot2","bot1","mid1"},{"top","mid1","bot1"}),
 "bot1" -> ({"mid1","top"},{"mid1"})]
```

The behaviour is promising. Now we may attempt to prove that ADD_MOD1 preserves the data type invariant of Xusage1:

Theorem 5.6: ADD_MOD1 is valid.

Proof: We must show that:

```
pre(xu,mod,uses) & inv(xu) & post(xu,mod,uses,xu') ==> inv(xu')
```

Suppose that the l.h.s. of the implication is TRUE. The proof then reduces to showing that for each module m in *dom* xu' the followings are TRUE:

```
(.A m1 £ xu'(m).u: m £ xu'(m1).b)                                  ---- (1)
```

and

```
(.A m1 £ xu'(m).b: m £ xu'(m1).u)                                  ---- (2)
```

Using *post*:

```
(.A m £ (dom xu - uses - {mod}): xu'(m) = xu(m))                   ---- (3)
```

and

```
(.A m £ (dom xu .I. uses): xu'(m).u = xu(m).u &
                           xu'(m).b = xu(m).b .U. {mod})
```

Now

```
                                                                   ---- (4)
(.A m £ (dom xu .I. uses): m £ xu'(mod).u)
<=>  (.A m £ (dom xu .I. uses): m £ uses)                          ---- (5)
```

Using (3),(4) and (5) the proof reduces to showing that (1) and (2) hold for any module m in uses .I. {mod}. We shall consider two separate cases:

(i) Let m = mod, (1) reduces to

```
(.A m1 £ uses: m £ xu'(m1).b)
```

This is obvious and immediate from *post*. Consider (2), the case when mod is recursive is obvious since (by *post*):

```
    mod £ xu'(mod).b
&   mod £ xu'(mod).u
```

So suppose mod is not recursive, i.e. ~(mod £ uses). Two cases must be considered: when ~(mod £ *dom* xu), by *post*:

```
xu'(mod).b = {}
```

hence (2) is immediate. When mod £ *dom* xu, by *post*:

```
xu'(mod).b = xu(mod).b
```

So (2) reduces to:

```
(.A m1 £ xu(mod).b: m £ xu'(m1).u)
```

which is immediate from *inv*-Xusage, (3) and (4).

(ii) Now consider `uses - dom xu`, we observe that (using *post*):

```
(.A m £ (uses - dom xu - {mod}): xu'(m).u = {} &
                                 xu'(m).b = {mod})
```

hence (1) is immediate, and (2) reduces to:

```
   (.A m £ (uses - dom xu - {mod}): (.A m1 £ {mod}:m £ xu'(m1).u))
<=> (.A m £ (uses - dom xu - {mod}): m £ xu'(mod).u)
<=> (.A m £ (uses - dom xu - {mod}: m £ uses)
```

which is immediate.

Theorem 5.7: ADD_MOD1 models ADD_MOD.

Proof: We must show two things:

(i) We must show that given `xu1 £ Xusage1` then

```
inv-Xusage1(xu1) & pre-ADD_MOD(retr(xu1),mod,uses)
==> pre-ADD_MOD1(xu1,mod,uses)
```

suppose that the l.h.s. of the implication is TRUE, and let:

```
xu = retr(xu1) = [m -> xu1(m).u: m £ dom xu1]          ---- (1)
```

Using *pre*-ADD_MOD:

```
~(mod £ dom xu) | xu(mod) = {}
```

Using (1) this reduces to:

```
~(mod £ dom xu1) | xu1(mod).u = {}
```

which verifies the r.h.s. of the implication.

(ii) Given that `xu1 £ Xusage1` we must show that:

```
inv-Xusage1(xu1) & pre-ADD_MOD1(xu1,mod,uses) &
post-ADD_MOD1(xu1,mod,uses,xu1') ==>
post-ADD_MOD(retr(xu1),mod,uses,retr(xu1'))
```

Suppose that the l.h.s. of the implication is TRUE. It is easy to show that `retr` is distributive over ++, i.e:

```
retr(m1 ++ m2) = retr(m1) ++ retr(m2)
```

for any two mappings m1 and m2 in Xusage1. Applying `retr` to *post*-ADD_MOD1 we get:

```
retr(xu1') = retr(xu1) ++ retr[m -> if m £ dom xu1 then
                                      mk-Cross(xu1(m).u, ...)
                                   else mk-Cross({}, ...)
```

```
                                  : m £ uses]
                ++ retr[mod -> mk-Cross(uses, ...)]
= retr(xu1) ++ [m -> if m £ dom retr(xu1) then
                                  xu1(m).u
                             else {}
                             : m £ uses]
                ++ [mod -> uses]
= retr(xu1) ++ [m -> xu1(m).u:
                             m £ (uses .I. dom retr(xu1))]
                ++ [m -> {}: m (uses - dom retr(xu1))]
                ++ [mod -> uses]
= (retr(xu1)++ [m -> retr(xu1(m)):
                             :m £ (uses .I. dom retr(xu1))])
  ++ [m -> {}: m £ (uses - dom retr(xu1))]
  ++ [mod -> uses]
= retr(xu1) ++ [m -> {}: m £ (uses - dom retr(xu1))]
                ++ [mod -> uses]
```

which verifies the r.h.s. of the implication and completes the proof.

From the theorems above it follows that operation ADD_MOD1 is correct and models ADD_MOD. The refinement, evaluation and verification of other operations is very similar to ADD_MOD and is not further discussed here. The complete specification of Xusage1 is given in figure 5.10.

```
ADT Xusage1
  DOM Xusage1 = Module -> Cross;
      Cross :: .u: Module-set, .b: Module-set;
  TYPE reaches: Module, Module, Xusage1 --> Bool;
  AUX inv-Xusage1(xu) ==
          (.A m £ dom xu: (.A m1 £ xu(m).u: m £ xu(m1).b) &
                          (.A m1 £ xu(m).b: m £ xu(m1).u));
  OPS
    INIT1: --> ;
      post(-,xu') == xu' = [];
    END INIT1

    ADD_MOD1: Module, Module-set --> ;
      pre(xu,mod,uses) == ~(mod £ dom xu) | xu(mod).u = {};
      post(xu,mod,uses,xu') == xu' =
          xu ++ [m -> if m £ dom xu then
                          mk-Cross(xu(m).u,xu(m).b .U. {mod})
                      else mk-Cross({},{mod})
                      : m £ uses]
             ++ [mod -> mk-Cross(uses,(if mod £ dom xu then
                                          xu(mod).b
                                       else {})    .U.
                                      (if mod £ uses then {mod}
                                       else {}))];
    END ADD_MOD1

    DEL_MOD1: Module --> ;
      pre(xu,mod) == mod £ dom xu;
      post(xu,mod,xu') ==
```

```
        xu' =   [m -> mk-Cross(xu(m).u - {mod}, xu(m).b - {mod})
                      : m £ (dom xu - {mod})];
    END DEL_MOD1

    USES1: Module --> Module-set;
       pre(xu,mod) == mod £ dom xu;
       post(xu,mod,-,ms) == ms = xu(mod).u;
    END USES1

    USED_BY1: Module --> Module-set;
      pre(xu,mod) == mod £ dom xu;
      post(xu,mod,-,ms) == ms = xu(mod).b;
    END USED_BY1

    REC_MOD1: --> Module-set;
      post(xu,-,ms) == ms = {m : m £ dom xu & reaches(m,m,xu)};

      pre-reaches(m1,m2,xu) == m1 £ dom xu & m2 £ dom xu;
      reaches(m1,m2,xu) == m2 £ xu(m1).u |
                           (.E m £ xu(m1).u: reaches(m,m2,xu));
    END REC_MOD1
END Xusage1
```

FIGURE 5.10 Specification of abstract data type Xusage1.

5.8 DISCUSSION

By borrowing ideas from the Vienna development method, we have arrived at a notation that, while preserving the useful features of VDM, such as conciseness and formality, lends itself to execution.

The provision of an abstract data type encapsulation mechanism on top of VDM has enabled us to formulate a software system specification at different levels of abstraction more easily. Two advantages are gained here. First, the encapsulation enables us to enforce useful disciplines, e.g. that an abstract data type is manipulated through its own set of private operations only. Second, it allows us to talk about abstract data types as objects, both conceptually and in reality. This in turn facilitates the construction of formal specification libraries which consist of self-contained abstract data type specifications. Libraries of this form would be an indispensable tool in software development and prototyping for two reasons. First, they significantly reduce the verification effort by allowing developers to build on top of each other's work; once an abstract data type is developed, verified and deposited in the library, subsequent users can employ it and rely on its correctness. Second, being executable, the library becomes a powerful tool for rapid prototyping. This is, of

course, the familiar reusable software approach to prototyping, but is more productive since it is applied to a higher, more stable level of abstraction.

Needless to say, by requiring our notation to be executable, we have necessitated some compromises concerning the implicitness of EPROL. For example, the implicit predicate:

```
(.E x £ Real: x**3 - 2*x = 2)
```

is not executable in EPROL since the search domain is potentially infinite. The implication of this is that certain styles of VDM predicates, while expressible in EPROL, are not executable. This does not necessarily mean that we have to restrict ourselves to executable constructs. Indeed, in our developments, we first produce a specification using any construct that we find appropriate. Once a specification is produced in this way, its transformation to an executable form is straightforward and involves very little effort (see chapter 9.)

The essential difference between the VDM approach and our approach is in the priorities these two assign to different aspects of development. VDM primarily concerns itself with rigorous verification of correctness from the start. EPROS regards verification as a complementary option; it primarily concerns itself with the appropriateness of a specification and with experimenting with alternative designs, and argues that executing a specification before verification can detect errors more easily and at a greatly reduced cost. This opinion is also shared by a number of other researchers [Goguen84, Kemmerer85, Henderson86a].

Chapter 6 IMPLEMENTATION

The better adapted a system is to a particular environment, the less adaptable it is to new environments.

- R A Fisher

This chapter describes the implementation notation of EPROL. It introduces various implementation constructs and the most basic facility for modularisation – imperative functions. Functions may be used for concrete realisation of several abstractions, e.g. abstract data types. Other forms of modules will be described in the next two chapters.

The implementation notation is not isolated from the specification notation. In fact, all the constructs and objects described in chapter 5 (e.g. combinators and sets) can be used freely in the implementation notation.

6.1 STATEMENTS

In the implementation notation, computation is usually defined in terms of statements. These are computation rules that cause useful side-effects.

assignment

This is the most elementary form of statement, the general form for which is:

```
location := expression
```

The effect of this statement is that `expression` is first evaluated and the resulting value is stored in `location`. Examples are:

```
x := 12**3;
ll := <<"x">,<"y","z">>;
m("John") := 30;
t.r := [1->1, 2->4];
```

where `x` is an integer, `ll` is a list, `m` is a mapping and `t` is a tree variable.

control structures

The simplest form of control statement is the if-then-else statement. It may take one of the following two forms:

```
if bool_expr then
   stat1;


if bool_expr then
   stat1
else
   stat2;
```

In both cases `bool_expr` is evaluated first. If it evaluates to TRUE then $stat_1$ is executed. If it evaluates to FALSE then, in the former case nothing will happen whereas in the latter case $stat_2$ will be executed.

There are two kinds of multi-branch control statements. The first is the mac statement ; this is very similar to the mac expression and has the general form:

```
mac {
   bool_expr1 => stat1;
   bool_expr2 => stat2;
    :
   bool_exprn => statn;
};
```

where `bool_exprs` to the left hand side are evaluated in the order they appear. If $bool\_expr_i$ evaluates to TRUE then $stat_i$ will be executed and the mac statement will terminate.

The second multi-branch statement is the cases statement; this is very similar to the cases expression and has the general form:

```
cases exprs {
   expr1 => stat1;
   expr2 => stat2;
    :
   exprn => statn;
};
```

where $expr_s$ is evaluated first and then $expr_i$s are evaluated in the order they appear. If $expr_i = expr_s$ then $stat_i$ will be executed and the cases statement will terminate.

loop structures

Three kinds of loop structure are provided. The for-do statement iterates over the elements of a set or a list. It has the general form:

```
for var in expr do
    stat;
```

where `var` is a bound variable and `expr` is a set or list expression. The bound variable needs no declaration. This statement iterates `var` over individual objects in `expr`. If `expr` is a set expression then iteration will be done pseudo non-deterministicly.

The while-do statement executes a statement repeatedly while a predicate is true, and has the general form:

```
while bool_expr do
      stat;
```

It evaluates the `bool_expr` first; if it evaluates to TRUE then it will execute `stat`. This process is repeated until `bool_expr` evaluates to FALSE at which time the loop is terminated.

The do-while statement executes a statement repeatedly until a predicate becomes false, and has the general form:

```
do
   stat;
while bool_expr;
```

Here `stat` is first executed, then `bool_expr` is evaluated; if it evaluates to TRUE then the whole process is repeated again, otherwise the loop is terminated.

Two other related statements are the `done` and the `goto` statement. The former appears in a loop structure and when executed terminates the loop immediately. The latter is used for explicit jump to a statement within a sequence of statements.

blocks

A sequence of statements may be grouped together to form a block by enclosing them within curly brackets, i.e.:

```
{ stat1;
  stat2;
   :
  statn;
}
```

A block is itself treated as a single statement.

assertions

Recording important invariants when writing programs is a good practice. In EPROL such invariants may be specified using assertion statements. An assertion statement has the general form:

```
assert(bool_expr);
```

When this statement is executed the `bool_expr` is evaluated; if it evaluates to TRUE then nothing will happen, otherwise the system will report that the assertion has failed. For example, the statement:

```
assert(.A i £ {1:len l - 1}: l[i] <= l[i+1]);
```

asserts that a list of numbers `l` is sorted in ascending order.

6.2 DATA TYPES

The data types described in chapter 5 (i.e. elementary types such as integers, and composite types such as sets) can be used directly in the implementation notation. In particular, trees may be used to implement record structures. Also, any abstract data types defined by the programmer in the specification part can be used freely.

Except for the elementary types, objects of all other types are dynamic. EPROL uses a heap storage mechanism for storing these objects, which is automatically garbage collected.

Certain additional data types are also provided, the use of which is restricted to the implementation part. These are briefly described below.

arrays

An array is a composite object of predefined length, containing a contiguous sequence of objects of the same type. For example,

```
array[5] Int
```

defines an array of 5 integers. Arrays may be one or multi-dimensional. In general, an array type has the following form:

```
array[e1][e2]...[en] Elem
```

where `e1`, `e2`, ..., `en` are arbitrary positive integer expressions, and `Elem` is the type of array

elements. Arrays may also be dynamic, i.e. their size may be decided at run time. Array elements are referenced in a manner identical to lists, albeit the index starts at 0.

files

A file refers to an external storage space; file is supported by a pre-defined type denoted by the keyword `file`. File operations are described in appendix C.3.

forms

Electronic forms are special abstract data types in EPROL (see chapter 7.) A form type is defined as: *`form`* `form_id`, where `form_id` is the name of a form module. Form operations are described in appendix C.4.

databases

The term database in EPROL refers to a collection of records which are stored and retrieved by a key. Every record in a database must be either a form image or a tree branch. For example,

```
DOM Student :: .name: Str, .age: Nat, .sub: Str;
    St_dbase = Student-dbase(key=name);
```

defines a database domain called `St_dbase` where every record in such a data base is an element of the domain `Student`. These records are stored by the `name` key. Similarly,

```
DOM Ap_dbase = (form Appliance_order)-dbase(key=$code);
```

defines a form database domain. Database operations are described in appendix C.5.

6.3 INPUT AND OUTPUT

Input and output can be performed with respect to standard channels, external channels, and windows. All such I/O is formatted. In addition, composite objects can be pretty printed using a special output function.

ordinary i/o

Standard and external formatted I/O is primarily supported by two functions called `put` and `get`. `Put` sends its output to either the standard output or an external file. It has the following call form:

```
put(file, format, arg1, ..., argn);
```

where `file` is optional and indicates the destination of output. `Format` is a string which contains the output format specifications for `arg1`, ..., `argn`. The format specifications follow the conventions of C [Kernighan78a].

`Get` obtains its input from either the standard input or an external file. The general call form for `get` is:

```
get(file, string, loc1, ..., locn);
```

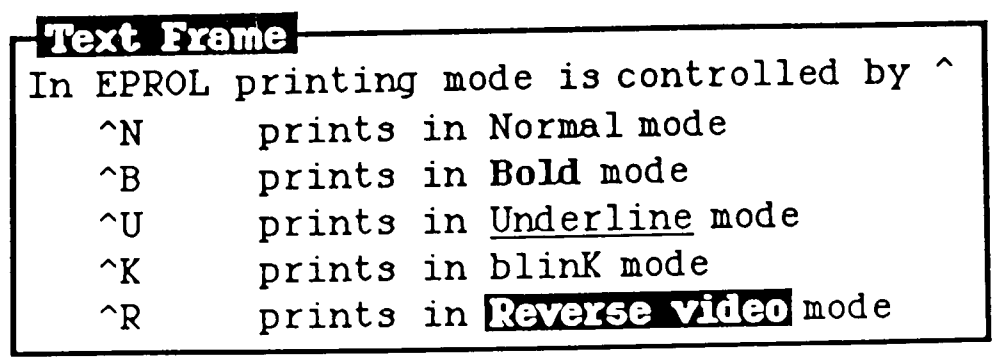
where `file` and `string` are optional. It first outputs `string` (if any and if `file` is not present) and then reads `n` values and stores them in locations `loc1`, ..., `locn`.

window-oriented i/o

EPROL supports the creation and manipulation of overlapping windows. Windows can be treated as channels for sending output and receiving input. Window functions and I/O operations are described in appendix C.4. An example of a window function is given below:

```
w_text(6,43,"^RText Frame^N",
       \In EPROL printing mode is controlled by ^^
            ^^N     prints in Normal mode
            ^^B     prints in ^BBold^N mode
            ^^U     prints in ^UUnderline^N mode
            ^^K     prints in ^KblinK^N mode
            ^^R     prints in ^RReverse video^N mode\);
```

It creates the following window:



pretty printing

The function `ppr` is responsible for pretty printing objects in EPROL and is extensively used by the interpreter. It takes an expression as argument and pretty prints its value. For example,

```
ppr(power{"apple","orange","pear"});
```

will produce:

```
{{},
 {"pear"},
 {"pear","orange"},
 {"orange"},
 {"pear","apple"},
 {"pear","orange","apple"},
 {"orange","apple"}
 {"apple"}}
```

6.4 IMPERATIVE FUNCTIONS

Imperative functions are defined by the FUNCTION module; they correspond to procedures and functions in modern programming languages and support procedural abstraction. Figure 6.1 shows the general structure of a FUNCTION module.

```
FUNCTION ifun_id (... parameter-list ...): result_type;
DOM ... private domain definitions ...
VAR ... private variable definitions ...

        :
      local definitions
        :
BEGIN
      :
    statements
      :
END ifun_id
```

FIGURE 6.1 The general structure of a FUNCTION module.

The parameter list and/or result type may be empty. It specifies the names and domains of function parameters. Each parameter is specified as

```
par: dom
```

if it is a value parameter, or as

```
VAR par: dom
```

if it is a variable parameter. The DOM part of a function is similar to the DOM part of an abstract data type. The VAR part defines one or more variables. For example,

```
VAR l: Int-list := <1,10,100>;
```

defines `l` to be a list of integers and initialise `l` to <1,10,100>.

The body of a function consists of a sequence of statements. A function can also contain local FUNCTION, DIALOGUE, FORM and CLUSTER modules. An example of a function module is given below. It is an implementation of the familiar quick sort algorithm.

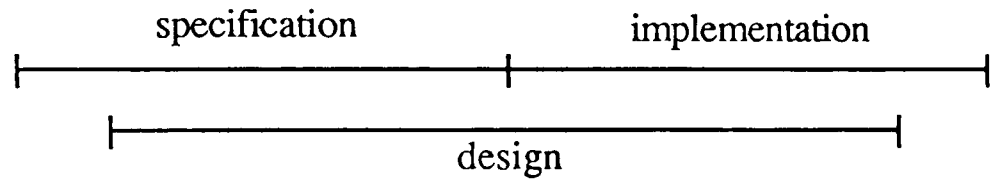
```
FUNCTION quick_sort(table: Table, size: Nat);
DOM Table = array[size] Str;
   FUNCTION quick_sort_aux(lower:Nat0, upper:Nat0);
   VAR i:  Nat0 := lower;
       j:  Nat0 := upper;
       key: Str := table[(lower+upper)/2];
      FUNCTION swap(VAR x: *, VAR y: *);
      VAR temp: *;
      BEGIN
          temp := x;
          x := y;
          y := temp;
      END swap
   BEGIN
       do {
          while table[j].name > key do j := j - 1;
          while table[i].name < key do i := i + 1;
          if i <= j then {
             swap(table[i],table[j]);
             i := i + 1;
             j := j - 1;
          };
       } while i <= j;

       if i < upper then quick_sort_aux(i,upper);
       if lower < j then quick_sort_aux(lower,j);
   END quick_sort_aux
BEGIN
     quick_sort_aux(0,size-1);
END quick_sort
```

6.5 DISCUSSION

The implementation notation described in this chapter fits next to the specification notation of chapter 5, and takes us down to the lowest level of abstraction. The provision of a sharp border between these two has ensured that the concepts would not be confused. This border, however, does not correspond to a sudden change of notation, but rather to

an extension of the notation.

Accepting the fact that design is a process of decision making, its separation from the above two is impossible; it therefore runs well into the specification and implementation notations. This is depicted by the following diagram.



So far we have been restricting ourselves to functionality only. This represents half of the picture in a software project. The other half, represented by the user interface, is of equal importance. It involves producing notations which support the specification and implementation of those parts of a system responsible for the dialogue between the user and the system. Our separate treatment of these two issues is a consequence of our desire to achieve such a separation both conceptually and in reality for reasons that were discussed earlier on.

The next chapter describes a notation for dialogue development which achieves such a separation. Our dialogue specification notation will be very different from that we used for specifying functionality. For implementation purposes, however, we shall be using the same notation as described in this chapter.

Chapter 7 USER INTERFACES

Software stands between the user and the machine.

- H D Mills

This chapter describes the notation of EPROL for user interface specification and development. This notation consists of an encapsulation mechanism for separating dialogues from functionality and a number of independent abstractions supporting well-developed concepts in user interfaces. Unlike functional specifications, the dialogue specification notation is initially graphical and semi-formal; this is described below.

7.1 STATE TRANSITION DIAGRAMS

The STD notation employed by EPROL is based on the one proposed by Denert [Denert77]. This is an extension of the usual STD notation and allows one to describe dialogue systems hierarchically. The symbols used in this notation are summarised in figure 7.1. Each symbol is briefly described below.

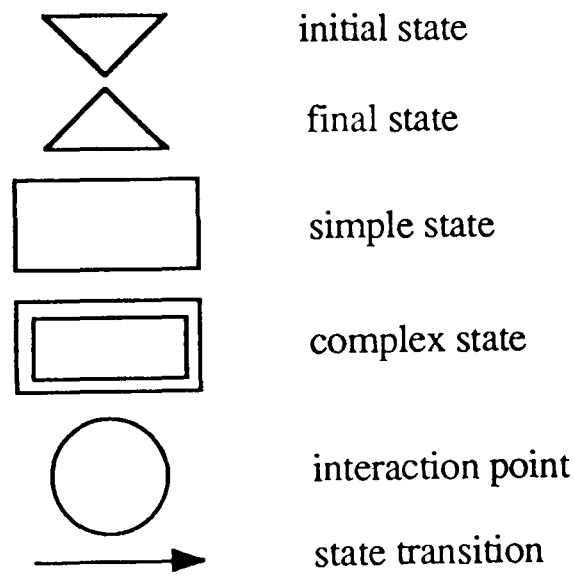


FIGURE 7.1 State transition diagram symbols.

- *Initial state* – Denotes the entry point for an STD. An STD must have exactly one initial state.
- *Final state* – Denotes the exit point for an STD. An STD must have exactly one final state.
- *Simple state* – Represents an action which involves no interaction with the user and is executed immediately. The action is usually described by a brief text in the box.

- *Complex state* – Is an abstraction of an entire STD which may be refined separately. A complex state may involve interaction with the user; for this reason, the system may remain in a complex state for an arbitrary length of time. A complex state may be labelled by a brief description of its function.
- *Interaction point* – Denotes a state in which actual interaction between the user and the computer takes place. Interaction points are usually labelled by a number or abbreviation.
- *State transition* - Indicates transition between states. Arrows entering and emerging from a complex state are conceptually tied to the initial and final states of the refinement of that state respectively. An arrow emerging from an interaction point must be labeled with user input or a predicate which will trigger that transition.

Complex states allow the abstraction of an entire STD in much the same way functions allow the abstraction of a sequence of processing steps. Using this notation, dialogue systems may be modularised and designed in a top-down manner.

the dialogue module

Once a dialogue system is specified as an STD it is then converted to the one dimensional notation of EPROL. This notation is supported by the DIALOGUE module. The general structure of a DIALOGUE module is shown in figure 7.2.

```
DIALOGUE dial_id(...parameter-list...): result-type;
DOM ...private domain definitions...
VAR ...private variable definitions...

                :
        local definitions
                :
BEGIN
    :
    state descriptions
    :
END dial_id
```

FIGURE 7.2 The general structure of a DIALOGUE module.

A DIALOGUE module represents a complex state. Each part of a DIALOGUE module is the same as that of a FUNCTION module except for the body. The body consists of one or more state descriptions where each state is either a simple state or an interaction point. A

simple state has the following form:

```
state state_id1: action => state_id2;
```

where `action` is a statement. It defines a simple state called $state\_id_1$ which performs the specified action and then moves to $state\_id_2$. $State\_id_2$ itself must be a simple state or interaction point in the same DIALOGUE module.

An interaction point has the following form:

```
iap stat_id: input_action;
           : pred1, output_action1 => stat_id1;
           : pred2, output_action2 => stat_id2;
           : pred3                 => stat_id3;
           : ....
           : ....
           : TRUE                  => stat_idn;
```

where `input_action` and `output_actions` are all statements. Each $stat\_id_i$ must be a simple state or interaction point in the same DIALOGUE module. Each predicate $pred_i$ is a predicate over user input or program variables. As shown above, `output_actions` are optional. Also, the last predicate may be simply TRUE, specifying a transition which will take place if no other predicate evaluates to TRUE. The above description defines an interaction point called `stat_id`. In this state, first `input_action` is performed; then the predicates are evaluated in the order they occur. If $pred_i$ evaluates to TRUE then $output\_action_i$ will be performed (if any) and the system will move to $stat\_id_i$. An `iap` description must specify at least one state transition.

The first state in a DIALOGUE body is assumed to be the initial state. The final state is defined implicitly using return statements. For example, in

```
state s1: action => return(10);
```

`action` is first executed and then the DIALOGUE module is terminated returning the number 10. All such return statements are conceptually tied to the DIALOGUE module's final state.

an example

This section will illustrate, by means of an example, the way in which a simple dialogue may be specified as an STD and then implemented by a DIALOGUE module.

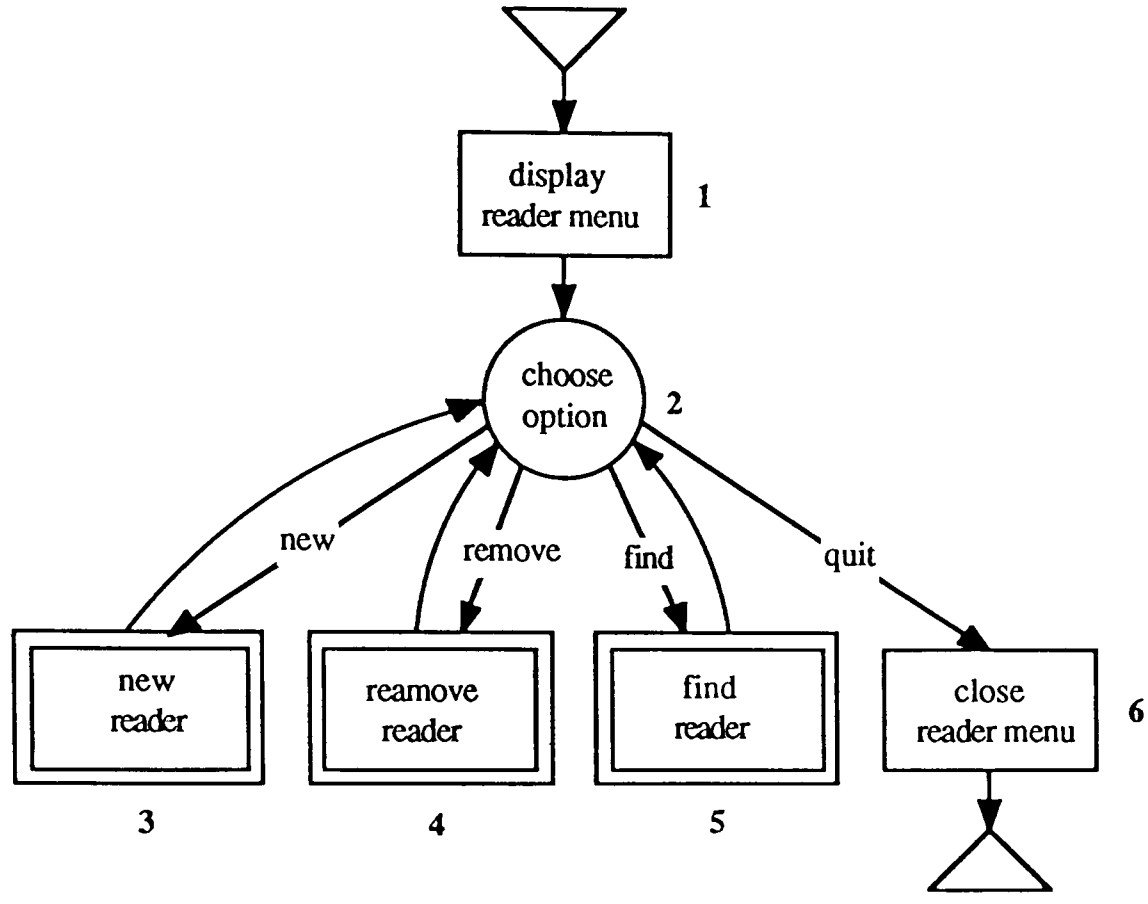


FIGURE 7.3 A simple state transition diagram.

Figure 7.3 shows an STD which is part of the user interface specification of a library system (see chapter 9.) The STD contains three complex states. A refinement of the complex state `remove reader` is shown in figure 7.4. The refinement is at the bottom of the dialogue hierarchy since it contains no further complex states.

The first state in the refinement is a simple state which creates a dialogue box within which all subsequent interaction will take place. The STD then moves to an interaction point which asks the user for a password. If the user types a wrong password the STD will move to state 4.3, report the error and move back to the interaction point. If the user makes too many wrong guesses the STD will move to state 4.5, warn the user that further interaction is denied and then move to state 4.10. When a correct password is given the STD will move to interaction point 4.5 where the user is asked for the id. of the reader who is to be removed. Invalid id's are handled by the simple state 4.6. Instead of giving an id., the user may quit the dialogue, in which case the STD will move to state 4.7, confirm the quit and then move to state 4.10. However, if the user supplies a valid id., the STD will move to state 4.8 and then 4.9, where it will remove the specified reader and confirm the removal respectively. The STD

will then move to state 4.10 where the dialogue box is closed, and lastly to the final state.

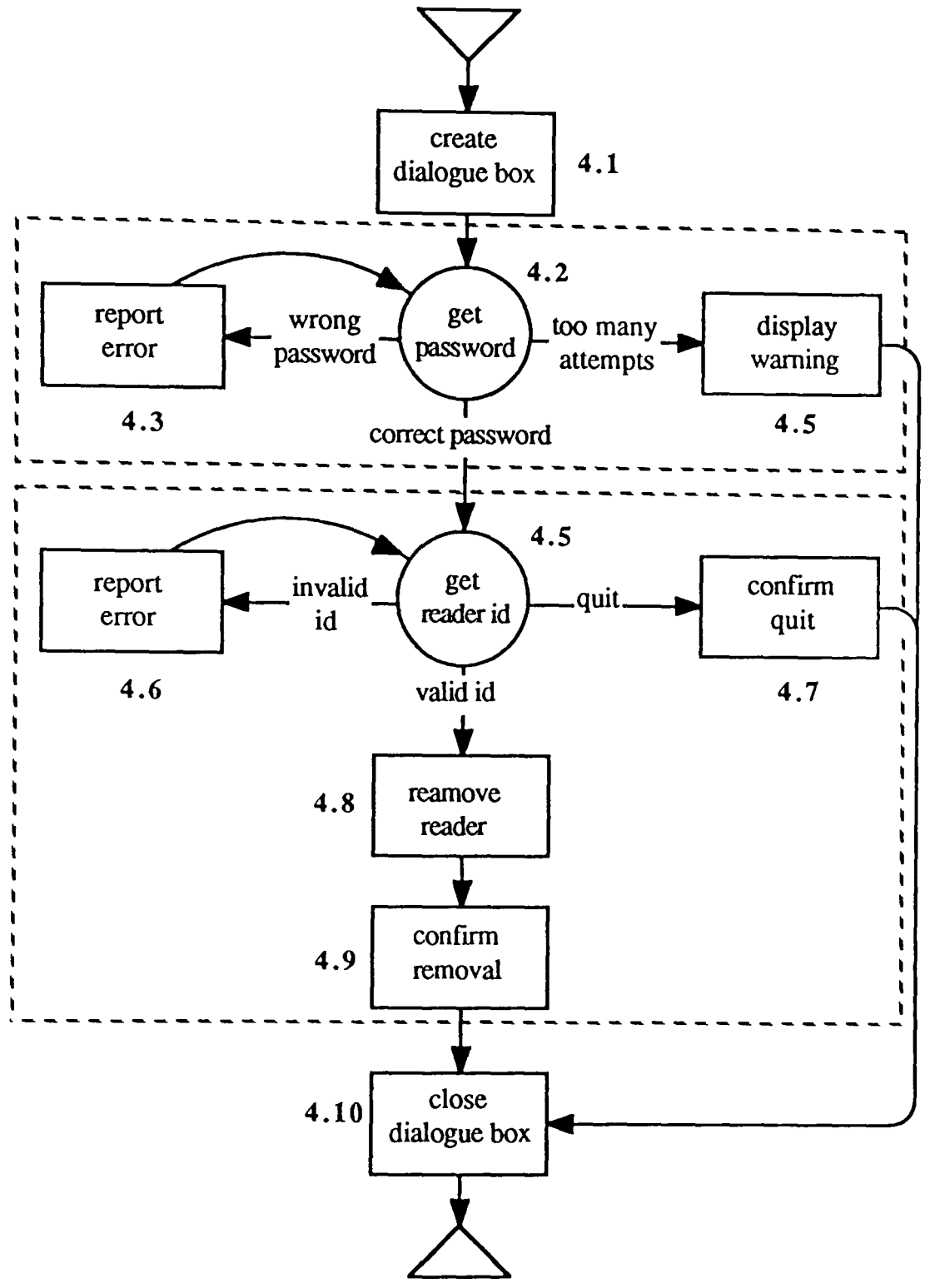


FIGURE 7.4 Refinement of complex state 'remove reader.'

The STD in figure 7.4 can be directly implemented by a DIALOGUE module. However, before implementation, one should always look for potential simplifications in the diagram. Typically, many simple states can be squeezed into their neighbouring interaction points. In this way one can reduce the number of states in a DIALOGUE module considerably and hence simplify the implementation. The two boxes with dashed lines in figure 7.4 depict this point. The first box, for example, suggests that simple states 4.3 and 4.5 can be squeezed into interaction point 4.2. This practice is referred to as *state reduction*.

An implementation of the STD, using the suggested state reductions, is given below. It consists of four states. Simple state box corresponds to state 4.1 in figure 7.4. The assertion in this state ensures that the user has permission to do a removal operation. Function message displays a note or warning at the bottom of the dialogue box. Interaction point pass corresponds to state 4.2 and its associated simple states. Interaction point read corresponds to state 4.5 and its associated simple states. The last state, out, is a simple state and corresponds to state 4.10.

```
DIALOGUE remove_reader(VAR rmv_list: Id-list, VAR rmv_ok: Bool);
VAR width:    Nat := 30;
    passwd:   Str;
    attempts: Nat0 := 0;
    id:       Id;
BEGIN
    state box: { assert(rmv_ok);
                 w_open(3,width, "^M Remove Reader ^N");
                 message(3,NOTE,"");
               }                                          => pass;

    iap pass: { w_move(1,1);
                w_get(" Password:  ",passwd,8,noecho);
                message(3,NOTE,"");
              };
              : passwd = DEL_PASS                         => read;
              : attempts >= ATTEMPT_LIM,
                { message(3,WARN,"Imposter!");
                  rmv_ok := FALSE;
                  wait(2);
                }                                         => out;
              : TRUE, { attempts := attempts+1;
                        message(3,WARN,"Wrong!");
                      }                                   => pass;

    iap read: { w_move(2,1);
                w_get(" Reader Id: ",id,5)
              };
              : db_find(rds_db,id) /= NIL,
                { rmv_list := rmv_list || <id>;
                  message(3,NOTE,"Removed");
                }                                                 => out;
              : id = 0, message(3,NOTE,"Quited")                  => out;
              : TRUE,   message(3,WARN,"Non-existant!")           => read;

    state out: w_close(1)  => return;
END remove_reader
```

Figure 7.5 shows the effect of the dialogue on the screen. It shows the dialogue box after a reader has been successfully removed.

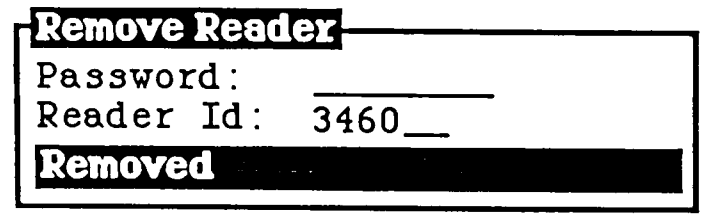


FIGURE 7.5 The dialogue box for removing a reader.

7.2 POP-UP MENUS

Many modern interactive systems are menu driven [Smith82a, Webster83]. In such systems the user interface usually consists of a network of menus where each menu serves a particular task. The user sends his request to the system by moving to the relevant menu and then selecting the required option.

Menus can be broadly classified into two categories. Each option of a menu in the first category depicts an action. Each option of a menu in the second category corresponds to a binary switch, i.e. it is either on or off. These two categories are supported by the menu and switch statements in EPROL respectively. Each is briefly described below.

the menu statement

A menu specification consists of the following:

- A menu title.
- A set of option names.
- A set of constraints where each option may be associated with at most one constraint. A constraint will indicate, at any point in time, whether an option is active. Options with no constraint are always active. Only active options may be selected by the user. The set of active options is called the *active set.*
- A set of actions where each option must be associated with one action.

Menus are specified by the menu statement; this has the general form:

```
menu {
   title
   option1, constraint pred1 => action1;
   option2                   => action2;
      :
      :
   optionk, constraint predk => actionk;
};
```

where `title` and `options` are all strings; `preds` are boolean expressions and `actions` are arbitrary statements. Options `1` and `k` above are both constrained; the second option is unconstrained.

As an example, consider a menu which allows the user to do insert, delete, and change operations on the records of a database. The menu specification will look something like this:

```
menu {
   "^RDB-operation^N"
   "Insert record", constraint size < MAX_SIZE => ins_rec();
   "Change record", constraint size > 0        => chg_rec();
   "Delete record", constraint size > 0        => del_rec();
   "Help" => menu {
                "^RHelp^N"
                "Insert" =>  .....;
                "Change" =>  .....;
                "Delete" =>  .....;
                "Back to last menu" => exit;
              };
};
```

Where modules `ins_rec`, `chg_rec` and `del_rec` deal with insertion, change and deletion of records and are not further specified here. The variable `size` depicts the number of records in the database. `MAX_SIZE` is an upper bound on the size of the database. The last option in the menu is unconstrained. The action associated with this option is itself a menu statement which provides help for operations in the original menu. The help texts are not specified here. The word `exit` in the last option of the help menu specifies that when this option is selected the help menu will be closed and control will be sent back to the original menu.

Figure 7.6a show what the menu will look like on the screen when actually activated. As shown in the figure, active options (i.e. 1 and 4) are printed in bold. The option the user is at (i.e. option 4) is always highlighted. The system ensures that the user will not be able to select inactive options. The user can move from the current option to the previous/next option by pressing the arrow keys, and selects an option by pressing the return key.

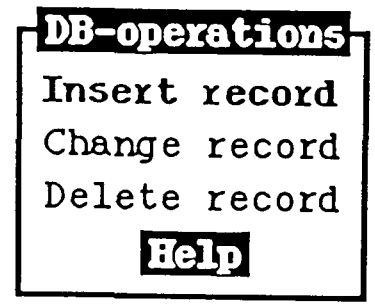


FIGURE 7.6a Menu as seen on the screen.

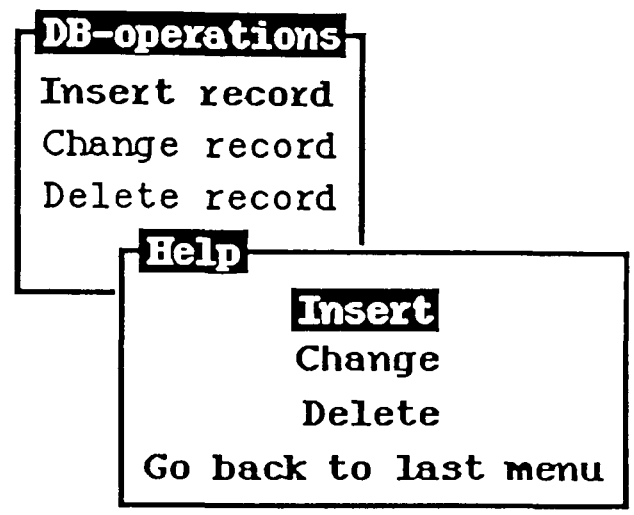


FIGURE 7.6b The help option is itself a menu.

Figure 7.6b shows the effect of selecting the last option. A further menu is opened, giving the options on which help is available.

the switch statement

A switch specification consists of the following:

- A switch title.
- A set of option names.
- A set of constraints as in a menu.
- A predicate per option. If this predicate evaluates to TRUE then the option will be set otherwise it will be reset.
- An action per option. The action is executed whenever the corresponding option is selected.

A switch statement takes the following general form:

```
switch {
    title
    option1, constraint cons1, tick pred1 => action1;
    option2,                   tick pred2 => action2;
```

```
        :
        :
    optionk, constraint consk, tick predk => actionk;
};
```

where constraints have the same role as they had in menus and are optional. Each option must be associated with a tick predicate. If this predicate evaluates to TRUE then the option will be ticked (i.e. marked on the left hand side to show that it is set). Like a menu, each switch option is associated with an action. When $option_i$ is selected $action_i$ will be executed.

To give an example, suppose we wish to allow the user to control the following parameters in a dialogue:

verbose	whether the system response should be brief or verbose.
warnings	whether the system should give warnings when it finds it appropriate to do so.
prompt	whether the dialogue prompt should be displayed or hidden.
cursor	whether the cursor should appear as a block or an underscore.
tabs	whether the system should convert tabs into spaces.

Each of these may be represented by a boolean variable, for example:

```
VAR verb, warn, prompt,
    blockcursor, tabconv: Bool := FALSE;
```

The facility may then be provided by a switch statement:

```
switch {
  "^RModes^N",
  "verbose",       constraint level>1, tick verb => verb := ~verb;
  "give warnings",tick verb | warn   => warn := ~warn | verb;
  "give prompt",  tick verb | prompt => prompt:= ~prompt | verb;
  "block cursor", tick blockcursor   => blockcursor := ~blockcursor;
  "convert tabs", tick convtabs      => convtabs := ~convtabs;
};
```

The variable `level` depicts the level of the dialogue. So, as specified in the first option, the user may only choose the verbose mode when he or she is at some level other than the first. The action for this option simply toggles the variable `verb`. The next two options are dependent on the first option, in the sense that when the dialogue is in the verbose mode the warning and prompt modes will be set anyway. This is ensured by including `verb` as a disjunction in the tick predicates of the second and third options.

Initially the switch frame will appear on the screen as shown in figure 7.7a. If the user, for example, selects the first option the first three options will be ticked, as shown in figure 7.7b. If the user again selects the first option the first three options will be reset taking us back to figure 7.7a.

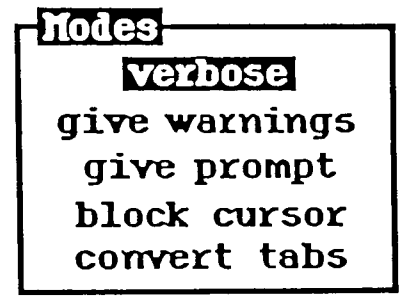


FIGURE 7.7a A switch frame.

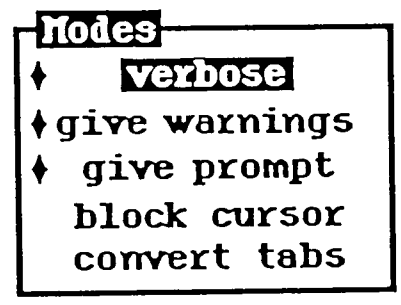


FIGURE 7.7b Switch frame after the first option is selected.

7.3 ELECTRONIC FORMS

A useful concept in user interface design are electronic forms. These are commonly used in office automation system and are exceptionally suitable for changeover from manual to computer-based systems [Tsichritzis80, Yao84]. The most useful aspect of forms is that they reflect the logical relationship between data items in a user interface [Tsichritzis82, Gehani82b].

the form module

The EPROL notation for defining electronic forms is based on the notation proposed by Gehani [Gehani83] who suggested that forms should be specified as abstract data types. Forms are defined using the FORM module; the general structure of which is shown in figure 7.8.

```
FORM form_id;

  \..................
   .. form layout ...
   ..................\

  :
  field definitions
  :
END form_id
```

FIGURE 7.8 The general structure of a FORM module.

The form layout part defines the layout of the form as it will appear on the screen. A form is always displayed in a window. The two backslash characters in the form layout depict the top left hand corner and the bottom right hand corner of the window. In a form layout, each field appears as a field identifier. This is just like a normal identifier, preceded by a $ symbol (e.g. $name).

Each field that appears in the layout part must be defined in the field definitions part. Each field is defined by specifying its type, maximum size, and optionally one or more attributes. The attributes define the properties of the field. An attribute may be one of the followings:

after	specifies that the field must be filled after certain other fields.
computed	gives a computation rule (a statement) that the system will use to compute the field automatically.
constraint	imposes a constraint that must be satisfied when the field is filled. If the constraint fails user data will be rejected.
initially	defines an initial value for the field. The field will maintain this value unless the user changes it during interaction.
lock	specifies one or more fields which will be locked after the field is filled.
noecho	specifies that the user data for the field must not be echoed on the screen (e.g. a password).
optional	specifies that the field is optional and may be filled if the user wishes to do so.
permanent	specifies that the field is permanent, i.e. once filled it may not be changed.
required	specifies that a value for the field is required and must be supplied by the user. This is the default case.
system	specifies that the field will be automatically filled by the system.

an example

To illustrate the use of the FORM module, consider the following form definition; it defines a form called appliance_order.

```
FORM appliance_order;
  \^BAppliance:^N
      Name:       $name                          Code:    $code
      Quantity:   $quantity                      Price:£ $price
                                           Total Price:£ $totprice
   ^BCustomer^N
      Name:       $cname
      Address:    $street
                  $town
                  $county
      Post Code:  $postcode                Delivery: $delivery

     Department: $dept                        Date: $date       \

  $name:      Str (12), required;      /* required is the default */
  $code:      Int (5), lock ($name);
  $quantity:  Int (5), constraint 0 < $quantity < 100;
  $price:     Real (6), after ($name,$code);
  $totprice:  Real (8), after ($quantity,$price),
              computed $totprice := $quantity*$price;
  $cname:     Str (14);
  $street:    Str (24);
  $town:      Str (24);
  $county:    Str (24);
  $postcode:  Str (7), optional;
  $delivery:  Str (4), after ($code),
              computed menu {
                        "By Post"          => $delivery := "POST";
                        "By Ship"          => $delivery := "SHIP";
                        "Special Delivery",constraint $code<=1000,
                                           => $delivery := "SDEL";
                        "To Be Collected" => $delivery := "TBC";
                       };
  $dept:      Str (12);
  $date:      Str (8), system (sdate);
END appliance_order
```

Everything in the form layout, apart from the field identifiers, is treated literally. Escape sequences are understood here too; for example, Appliance and Customer are both specified to be printed in bold.

In the field definition part a field is typically defined to be of type Int, Real or Str. The first field, for example, is defined to be of type Str having a maximum length of 12 characters. This field is also defined to be required. The second field has a lock attribute; it

specifies that when it is filled the `$name` field will be locked. The third field specifies, by means of a `constraint` attribute, that `$quantity` must be an integer between 0 and 100. The fifth field is computed automatically; here, an `after` attribute is used to ensure that all variables used in the computation are already bound. The last field is filled automatically by the system; the identifier `sdate` here stands for short date (e.g. `12/02/86`).

Figure 7.9a shows what the form will look like on the screen when activated. As shown there, the position of the fields directly conforms to that specified by the form layout part. Figure 7.9b shows the form when the user is actually filling the `Delivery` field. This was defined to be a `computed` field where computation is performed by a menu.

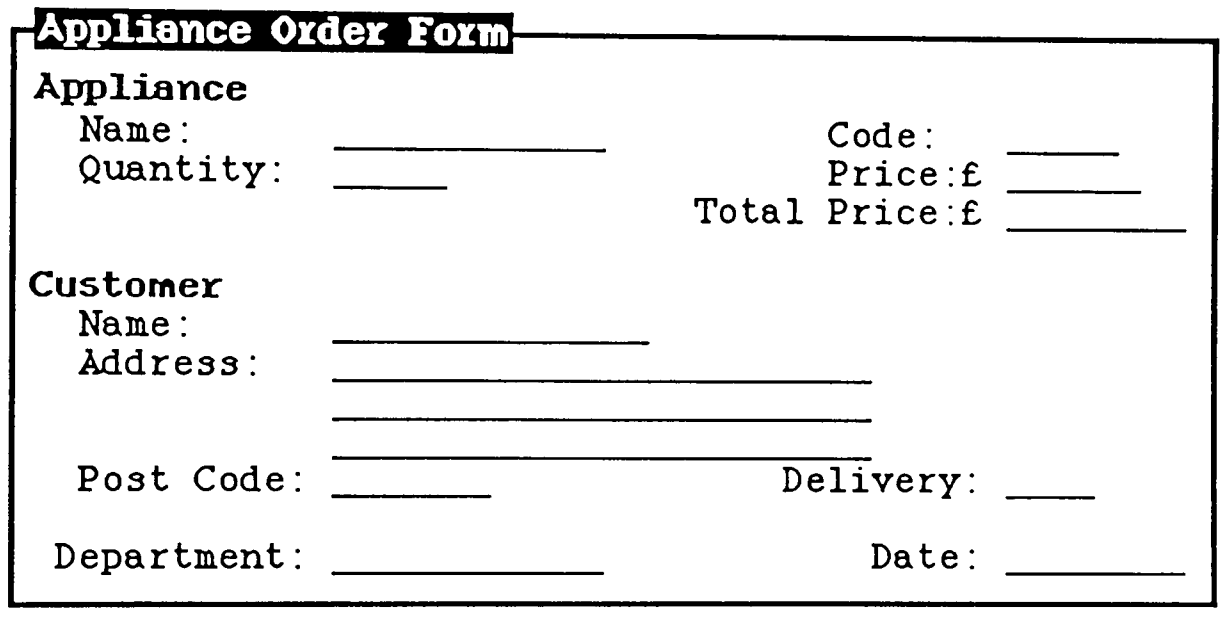


FIGURE 7.9a Form as seen on the screen.

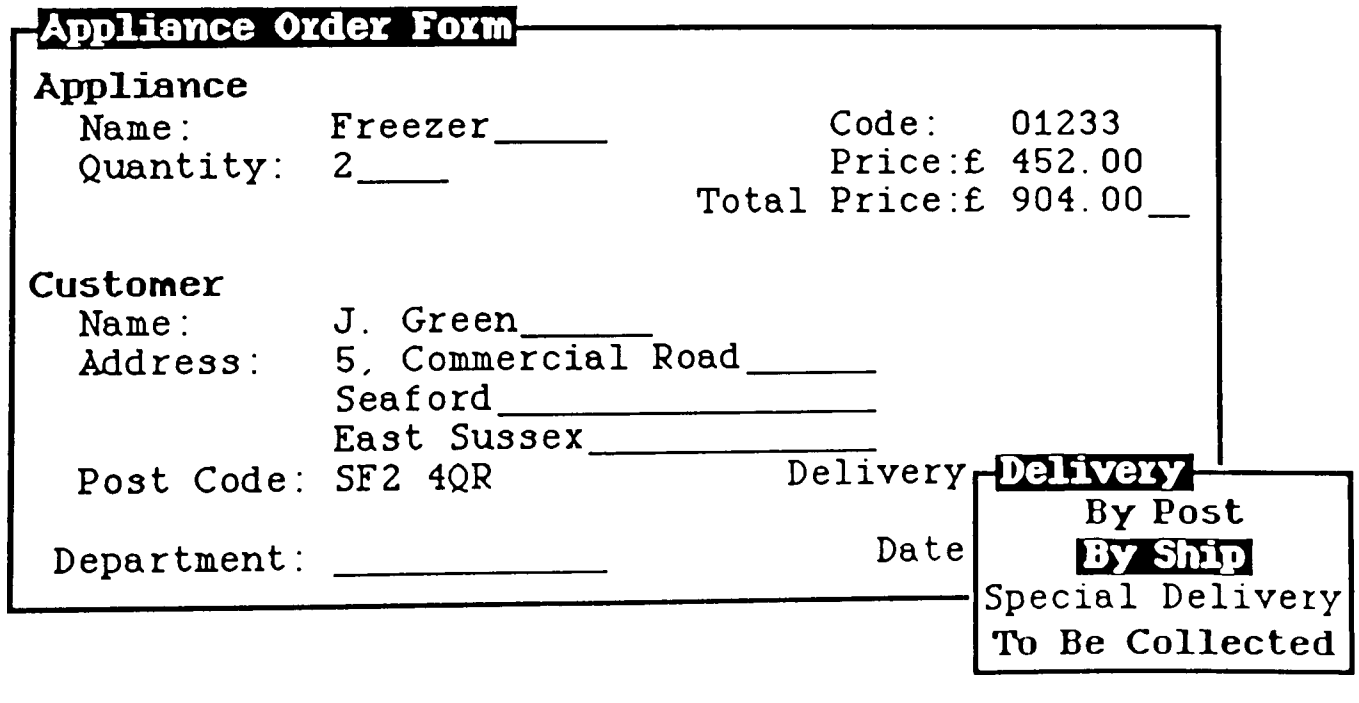


FIGURE 7.9b Delivery field is *computed* and menu driven.

The user fills a form by using the arrow keys to move to previous/next field. No particular order is imposed on the way a form may be filled other than that specified by the attributes. Other function keys may be used to cancel a field/all fields, exit from the form, quit the form, get help from EPROS, etc. The system performs many checks on user actions to ensure correctness. One such check, for example, concerns the type of data. An example is shown in figure 7.10a where the user attempts to assign a non-integer value to the `Quantity` field. In this case, the error frame will last for a short while on the screen and will disappears automatically. The field will be then cleared to allow the user to re-enter the data. Other checks ensure that the facts specified by field attributes remain integral. An example of this is shown in figure 7.10b where the user attempts to fill the `Price` field before the `Code` field.

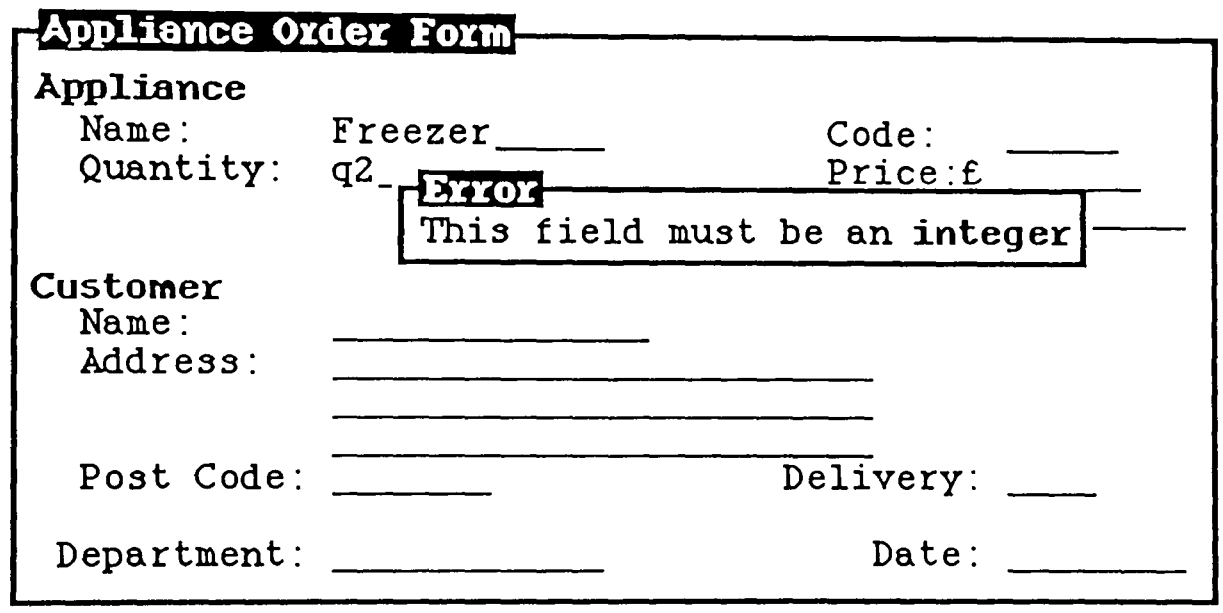


FIGURE 7.10a Example of a type error.

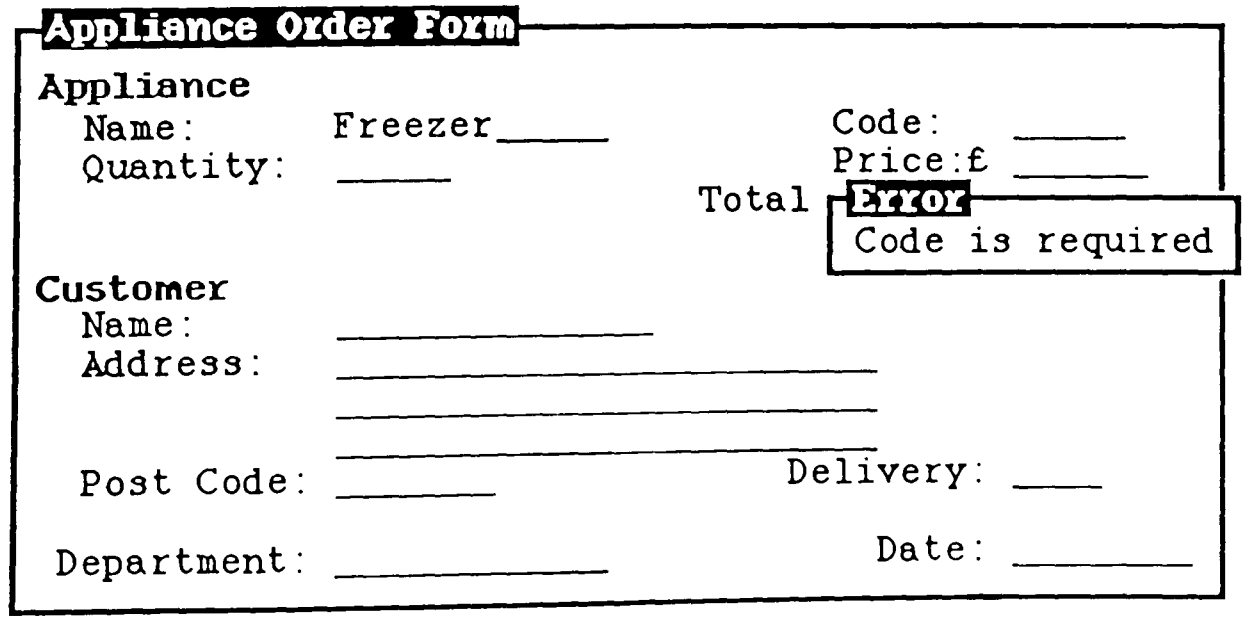


FIGURE 7.10b Example of an attribute violation.

Once a form is filled the user may complete the task by pressing the EXIT key. The system will then check all the fields to ensure that everything is in order (for example that all non-optional fields have been filled.) If not, it will give appropriate messages to guide the user in completing the form.

7.4 DISCUSSION

The ability to separate a dialogue from the usual processing in a program is an important one. For one thing, the dialogue part stands out, explicitly indicating where and how it fits with the rest of the system. As a result, it simplifies difficult tasks such as changing the user interface to a system and introducing multiple interfaces to the same system. Also, it encourages the developer to think of the user interface as an entity separate from the rest of the system.

The dialogue specification and development notation described in this chapter enables us to achieve such a separation. As we saw, the modularisation concept is a direct extension of the familiar notion of procedural abstraction and supports hierarchical development in a similar way.

Our notation is strongly based on the STD concept and regards each separate dialogue as consisting of individual states connected through transitions which are invoked by predicates over user input and system states. Obviously, such a framework can also be represented by the implementation notation, where each state transition is realised by a goto statement. By restricting ourselves to a specific and tighter notation, however, we have gained the advantage of imposing a discipline which directly reflects the STD concept. The indication that a state is simple, complex, or an interaction point, for example, has on its own, enhanced the readability of dialogue specifications and has increased the amount of information that can be conveyed by a dialogue description.

We also showed how other self-contained abstractions can be exceptionally useful in dialogue development, and how they can lead to the specification and direct execution of certain interactions, rather than their time-consuming implementation. A question that arises at this point is how and what other abstractions may be useful in dialogue design. This is a

difficult question and can be properly answered only in the light of extensive experience. A useful criterion that we have used in this respect, and which has proved effective, is that concepts which are used repeatedly and which can be generalised should be abstracted. The provision of menus and forms, for example, reflects the use of this criterion. However, following this line of abstraction is not easy unless we have a higher order abstraction facility which allows us to *design* such abstractions with considerable ease and without disturbing the base language. This brings us to the concept of clusters and meta abstraction which is the topic of the next chapter.

Chapter 8 CLUSTERS AND META ABSTRACTION

Everything should be made as simple as possible, but no simpler.
- A Einstein

Two important techniques of abstraction, that is *data abstraction* and *procedural abstraction*, have already been discussed. Data abstraction was extensively covered in our discussion on abstract data types. Procedural abstraction was described in the context of FUNCTION modules.

This chapter returns back to the topic of procedural abstraction to introduce a new and novel abstraction technique called *cluster*. Clusters may be regarded as a generalisation of current techniques for procedural abstraction and are particularly useful in situations where procedures and functions are inadequate, and unable to capture the required level of abstraction.

8.1 THE NEED FOR CLUSTERS

Clusters, in fact, have already been used in this thesis. The menu and switch statements described in chapter 7 are two good examples; these are predefined clusters in EPROL. To justify the need for clusters, we shall go back to the problem of specifying menus and consider the difficulties that we may encounter when we attempt to realise menus using FUNCTIONS.

As stated in chapter 7, a menu specification consists of the following:

- A menu title.
- A set of option names.
- A set of constraints, each associated with an option.
- A set of actions, one per option.

The first problem we encounter is that the number of data items is by nature variable. As a result, the data has to be passed to a function using composite data structures. Lists seem to be a good choice. Consider the following function:

```
FUNCTION menu(title: Str, options: Option-list): Choice;
DOM Option :: .op_name: Str, .cons: Bool;
```

```
    Choice = Nat;
BEGIN
 /* draw the menu.
    print the title.
    print the options:
          if an option is active print it in bold
          otherwise print it in normal mode.
    loop forever:
          cases pressed-key {
                up-arrow:   move to previous option.
                down-arrow: move to next option.
                return-key: return the option number.
          }
 */
END menu
```

Every object in the domain `Option` consists of an option name and its constraint. The function returns a unique id. in the domain `Choice` which identifies the selected option. A sample call to this function is shown below:

```
cases menu ("TEST", <mk-Option("option1",pred1),
                     mk-Option("option2",pred2),
                      :
                     mk-Option("optionk",predk)>) {
 1: action1;
 2: action2;
  :
 k: actionk;
};
```

Although this approach works it has two drawbacks:

- The association of options and actions is controlled *outside* the menu function. Each call requires an additional cases statement to manage this. As a result, each call is longer and more complicated than it should be. Furthermore, this increases the possibility of introducing some inconsistency between options and actions. For example, suppose that during maintenance a new option is inserted in the middle of the option list. This will require a consistent re-numbering of the `cases` branches and is potentially error prone.
- The function will not allow the user to select more than one option from a menu. For example, the user cannot select option 2 then option 6 and so on. To do so, one will have to call the function repeatedly. This is unreasonable since it will display the menu everytime the function is called whereas one display would be sufficient. Note that repeated calls cannot be avoided since the constraints are first evaluated and then passed as booleans. Because the active set may change during execution, passing the constraints after each action execution is essential.

Both these problems can be avoided by passing the constraints and actions symbolically (i.e. in an unevaluated form), and managing them *inside* the menu module itself. However, function parameters are not powerful enough to support this.

8.2 THE CLUSTER MODULE

The cluster module has been especially designed to avoid the kind of problems mentioned in the previous section. The general structure of a cluster module is shown in figure 8.1.

```
CLUSTER clus_id {...cluster-scheme...};
DOM ...private domain definitions...
VAR ...private variable definitions...

        :
      local definitions
        :
BEGIN
    :
    statements
    :
END clus_id
```

FIGURE 8.1 The general structure of a CLUSTER module.

A cluster definition consists of three distinct parts. These are cluster scheme, local definitions, and cluster body. A cluster scheme is a syntactic description embedded with semantic descriptions such as type of objects, and effectively defines the syntactic domain of a cluster. It is composed of syntax operators and objects with pre-defined syntax and semantics.

The domain, variable and local definitions parts are identical to that of functions. Cluster modules may be nested in exactly the same way as other modules such as functions and dialogues. A cluster body is also very similar to that of a function; it consists of a sequence of statements.

A cluster scheme is defined using a meta notation which allows the definition of syntactic rules to describe the way in which objects may be grouped, ordered and related to each other. This notation is described below.

the meta notation

The meta notation is very similar to the Backus-Naur Form (BNF) notation [Naur63] used for specifying the syntax of programming languages. The notation is based on a number of meta characters.

The characters { and } are used to specify repetition. Objects appearing between { } may be repeated a number of times. The characters [and] specify optional objects. Any object (or group of objects) appearing between [ ] is considered to be optional. The characters (and) are used for grouping and to override the precedence of other meta characters. A vertical bar | will specify choice from a group of objects. Characters * and + may be used in association with { } to specify zero or more, and one or more appearances respectively. Finally, single quotes ' ' are used to specify literals. Literals are arbitrary sequences of characters.

For example,

```
{object}*n
```

specifies that `object` may appear zero or more times and that the number of appearances will be recorded in variable `n`. Similarly,

```
[object]n
```

specifies that `object` may or may not be present; `n` will be one if it is present and zero if not. An example of using the choice character | is:

```
(object1 | object2 | object3)n
```

where one of $\texttt{object}_1$, $\texttt{object}_2$ or $\texttt{object}_3$ must be present; `n` will be 1, 2 or 3 indicating which one is present. Variable `n`, used in the above examples, is called an *indicator*; it records a specific instance of a meta expression.

A cluster scheme is a meta expression and is composed of meta characters and four predefined object classes. The object classes are constants, expressions, statements and identifiers, represented by the keywords `Const`, `Exprn`, `Statm` and `Ident` respectively. The exact syntax and semantics of these is that established by EPROL itself. A short informal description of each is given below:

`Exprn`	a composition of variables, constants, operators, functions, etc. which when evaluated produces a value. These usually do not cause any side-effects.
`Const`	an `Exprn` which can be, and is, evaluated at compile time.
`Statm`	computation rules which achieve their ends through useful side-effects.
`Ident`	a unique sequence of alphanumeric characters. Examples are variable and function names.

To avoid confusion, we should stress that literals and identifiers are totally different things. Literals have very simple semantics - they map to themselves - whereas identifiers represent objects with more elaborate meaning (e.g. a variable name). The meta notation is summarised in figure 8.2.

`{object}k`	- `object` must appear k times exactly.
`{object}*n`	- `object` may appear zero or more times.
`{object}+n`	- `object` may appear one or more times.
`[object]n`	- `object` is optional.
`(object)`	- `object` itself, useful for grouping.
`(object1 | object2 | object3)n`	- Exactly one of `object1`, `object2` or `object3` must be present.
`'charseq'`	- charseq is a literal and maps to itself. Other examples are `'=>'`, `','` and `'constraint'`.
`Exprn`	- expression.
`Const`	- constant.
`Statm`	- statement.
`Ident`	- identifier.

where k *is a positive integer and* n *is an indicator.*

FIGURE 8.2 Summary of the meta notation.

Comparing the four object classes just described to parameters in a function, we observe a few differences: unlike parameters, objects are generalised, syntax directed, and may be symbolic (as opposed to a value). For this reason, we shall use the term *object* to refer to them hereafter. Similarly the terms *actual object* and *formal object* will be used in the place of actual and formal parameter.

cluster schemes

In a cluster scheme, an object is specified by a unique name followed by an object class followed by a type specification (if required). For example,

```
x: Exprn: Real
```

specifies x to be an object in the object class Exprn having the type Real. All objects require a type specification except Statm for which the type is always void, e.g.:

```
s: Statm
```

specifies s to be in the object class Statm. The object class Ident can have the most general type specification. For example,

```
id: Ident: Nat --> Nat-list
```

specifies id to be in the object class Ident and having a function type clause which maps natural numbers to lists of natural numbers.

The meta notation provides a powerful means of grouping objects together with considerable ease. The followings are two simple examples of its use:

```
'if' cond: Exprn:Bool 'then' st1: Statm
                    ['else' st2: Statm]n ';'

'begin' {st: Statm ';'}+n 'end'
```

The first example specifies an if-then-else statement where the else part is optional. The second example specifies a Pascal-like begin...end compound statement. In the second example, the object st occurs within { } and automatically becomes a list of statements. The length of this list is indicated by the value of the indicator n. So, for example, in

```
begin
    i := i+1;
    k := k-1;
    f(i,k);
end;
```

st becomes a list of three statements, i.e.:

```
st = <i := i+1, k := k-1, f(i,k)>
```

Individual statements may be accessed by indexing the list; i.e. st[1], st[2] and st[3]. In general, every level of nesting by { } makes the formal objects within the nesting a list of whatever they are. So in

```
{...{...{... ex: Exprn:Int ...}*k ...}+m ...}+n
```

the formal object st is a list of lists of lists of integer expressions, i.e.:

```
ex: Int-list-list-list
```

The same rule equally applies to indicators. So, for example, n, m and k are of types:

```
n: Int
m: Int-list
k: Int-list-list
```

Such types are automatically setup by the EPROL compiler. Note that [] does not produce any nesting effects. For example, in

```
[...{... ex: Exprn:Int ...}+k ... es: Statm ...]*m
```

the types are:

```
m:  Int
es: void
k:  Int
ex: Int-list
```

8.3 A CLUSTER DEFINITION

To illustrate the use of clusters we shall define a variant of the menu statement of EPROL as a cluster. This definition is useful in the sense that it shows how various parts of a cluster relate to each other. In particular, it shows how formal objects and indicators are manipulated. The complete definition is given below:

```
%library "scr"
%library "str"

CLUSTER menu { 'title' title: Const:Str
               { 'option' optn: Const:Str
                 [',' 'constraint' cons: Exprn:Bool]m
                 '=>' action: Statm ';'
               }+n
             };
VAR  active:  array[n] Bool;
     margin:  array[n] Nat0;
     max_len: Nat0 := 0;
     cur_opn: Nat0 := 1;
     opn_len: Nat0;

     FUNCTION update_active_set ();
     VAR  actv: Bool;
     BEGIN
         for i in {1:n} do {
             actv := m[i]=0 | cons[i];
             if (active[i] /= actv) then {
                active[i] := actv;
                w_move (i, margin[i]);
                w_put ("%s%s",if actv then "^B"
                              else "^N", optn[i]);
```

```
            };
        };
    END update_active_set

BEGIN
    for i in {1:n} do {
        opn_len := st_len (optn[i]);
        if (opn_len > max_len) then
           max_len := opn_len;
    };
    w_open (n, max_len, title);
    for i in {1:n} do {
        margin[i] := (max_len - st_len (optn[i])) / 2 + 1;
        active[i] := m[i]=0 | cons[i];
        w_move (i, margin[i]);
        w_put ("%s%s", if active[i] then "^B"
                       else "^N", optn[i]);
    };
    while TRUE do {
        w_move (cur_opn, margin[cur_opn]);
        w_put ("%s%s", if active[cur_opn] then "^M"
                       else "^R", optn[cur_opn]);
        w_move (cur_opn, 1);
        cases keybd () {
           'F1' => { w_move (cur_opn, margin[cur_opn]); /* next */
                     w_put ("%s%s", if active[cur_opn] then "^B"
                                    else "^N", optn[cur_opn]);
                     cur_opn := if cur_opn = n then 1
                                else cur_opn+1;
                     };
           'F2' => { w_move (cur_opn, margin[cur_opn]);
                     w_put ("%s%s", if active[cur_opn] then "^B"
                                    else "^N", optn[cur_opn]);
                     cur_opn := if cur_opn = 1 then n /* previous*/
                                else cur_opn-1;
                     };
           'F3' => if active[cur_opn] then {
                     action[cur_opn];                  /* select */

                     update_active_set ();
                   }
                   else bell ();
           'F4' => done;                               /* exit */
           'F5' => w_text (5, 30, "^RMenu-help^N",     /* help */
                                  \^BF1^N - go to next option
                                   ^BF2^N - go to previous option
                                   ^BF3^N - select this option
                                   ^BF4^N - quit this menu
                                   ^BF5^N - this help\ );

           TRUE => bell ();
        };
    };
    on_exit do
            w_close (1);
END menu
```

The definition makes use of two standard libraries called `scr`, for screen management, and

`str`, for string manipulation (see appendix C.) The cluster scheme is the part appearing between curly brackets just after the cluster id.; `title`, `option`, `,`, `constraint`, `=>` and `;` are all literals. The cluster scheme contains four named objects; these are `title`, `optn`, `cons` and `action`. The first object, `title`, is a string constant. The second object is a list of string constants since it occurs inside a repetition. The third object is a list of boolean expressions, and the fourth object is a list of statements. The definition also contains two indicators; `n` is an integer and records the number of options etc.; `m` is an integer list indicating which options have constraints.

The local variable definition part defines two dynamic arrays called `active` and `margin` of types boolean and positive integer respectively. Note how the indicator `n` is used to specify the dimension of these arrays. Array `active` indicates which option is active at any time. Array `margin` records a left margin for each option so that it may be printed in the centre. The local function `update_active_set` updates the active set of the menu after each action execution.

The first loop in the cluster body finds the maximum length of options and records it in `max_len`. A window is then opened which is `n` lines long and `max_len` characters wide, having the title `title`. The next loop prints the options in this window, centring each option on a line and printing active options in bold.

The last loop executes user commands. Each time round the loop, the current option is highlighted on the screen; it is printed in mixed mode if active and in reverse video if inactive. A cases statement is used to decide which key is pressed by the user: `F1` moves to the next option, `F2` moves to the previous option, `F3` selects an option, `F4` exits from the loop, and `F5` produces a help frame. Any other key is rejected by ringing the margin bell. Also note that `F1` and `F2` produce a wrap around effect when the user is at the last or the first option respectively.

Note that every reference to an `Exprn` or `Statm` object causes evaluation of that object at run time. For example, `cons[i]` evaluates and returns the value of the `i`-th constraint. `Const` objects, on the other hand, are evaluated at compile time. It follows, therefore, that `Const` objects can be arbitrary expressions which do not refer to any free variables. For

example , `title` is a `Const` object and in an actual call it may be

```
st_conc ("Menu ", "2.5")
```

where `st_conc` is a string concatenation function. This expression is evaluated at compile time and is replaced by the constant `"Menu 2.5"`.

An example of a call to the menu cluster is shown below. It has the same effect as the one corresponding to figure 7.5 in chapter 7. The only difference is that this call contains two more literals (i.e. `title` and `option`) and that is because of the way we have defined our cluster.

```
menu {
  title  "^RDB-operation^N"
  option "Insert record", constraint size < MAX_SIZE => ins_rec();
  option "Change record", constraint size > 0        => chg_rec();
  option "Delete record", constraint size > 0        => del_rec();
  option "Help" => menu {
                     title  "^RHelp^N"
                     option "Insert" =>  .....;
                     option "Change" =>  .....;
                     option "Delete" =>  .....;
                     option "Back to last menu" => exit;
                   };
```

8.4 TERMINATION MECHANISMS

There are four ways in which a cluster may be terminated. These are:

- By a static `return` statement in the cluster body.
- By a dynamic `return` statement in a cluster call.
- By a dynamic `exit` statement in a cluster call.
- By flow of control reaching the end of the cluster body.

Often before returning from a cluster we would like to ensure that certain tasks are properly terminated. For example, in our menu cluster, we must ensure that the menu window is closed before exiting from the cluster. One way to achieve this is to require each return statement to be preceded by a `w_close(1)` statement. However, such a solution is very unwise as it exposes a major design decision to the user and places considerable burden upon him. An alternative approach, offered by the cluster mechanism, is to use an `on_exit do` statement. This specifies a statement which is always executed before leaving the cluster.

To illustrate the features of the termination mechanism consider the following example. It is a partially defined function which contains a nested call to the menu cluster.

```
FUNCTION foo(): Int;
BEGIN
   :
   menu {
     title  "^RDB-operation^N"
      :
     option "Help" => menu {
                        title "^RHelp^N"
                        option "Insert" =>  .....;
                        option "Change" =>  .....;
                        option "Delete" =>  .....;
                        option "Quit this menu"     => exit;
                        option "Quit previous menu" => return(0);
                      };
   };
   :
END foo
```

The `exit` statement in the above example terminates the inner call. This causes the `on_exit do` statement for the inner menu call to be executed. Hence the window of this menu will be closed and control will be transferred to the outer menu. This is an example of a dynamic `exit` statement. The `return` statement above is static with respect to function `foo`, and dynamic with respect to both menu calls. When executed it first causes the `on_exit do` statement of the inner menu to be executed and then the `on_exit do` statement of the outer menu call. Therefore, both menu windows will be closed successively. Then function `foo` will be then terminated and the value 0 will be returned as the result of the function.

As a general rule, therefore, it can be stated that:

- A dynamic `exit` terminates the most recently invoked cluster (which is still active).
- A dynamic `return` terminates all clusters in a nested call (which are still active) until a module body is reached.

8.5 APPLICATIONS OF CLUSTERS

The most important use of clusters is for modular software design. Two advantages may be gained here. Firstly, the notational power of clusters simplifies the task of properly decomposing a system into modules according to the important criteria laid down by Parnas [Parnas72, Parnas79]. This is because clusters have a far greater potential for information

hiding than functions. For example, in the function version of menu we had to expose a major design decision to the user and require him to manage the association of options and actions outside the function. This decision was properly hidden by the cluster version which managed the association inside the cluster.

Secondly, clusters facilitate the construction of truly reusable software modules. The primary reason for this is that, unlike functions which are based on rigid interfaces, clusters allow the programmer to program the interface. In this way one can cater for a variety of call requirements without exposing any internal details of a module.

A further use of clusters is for pseudo language extension. Using this approach, a number of constructs may be added to the base language to support and simplify the task of implementing specific applications. An interesting area here is user interface design, where clusters may be used for designing dialogue facilities as abstractions. One general abstraction of this kind is what we call *dialogue box* and is described below.

dialogue boxes

In window-oriented user interfaces usually all dialogue takes place within windows. Earlier on, we saw two styles of such windows (i.e. menus and forms.) A further style is what might be called a dialogue box. A dialogue box has some similarity to a menu or a form in the sense that it embodies a dialogue with a predefined protocol. Unlike menus and forms, however, the protocol is controlled by the programmer and may vary considerably from one dialogue box to another.

We illustrate the concept by an example. The following is a dialogue box definition taken from a library system which will be described in chapter 9. The corresponding dialogue box frame is shown in figure 8.3.

```
dial_box {
    "^M Find Book ^N"
    field " Code: ",    code: 6,  empty 0 => commands;
    field " Author: ",  auth: 20, empty "";
    field " Title: ",  title: 25, empty "";
    command " FIND " => { books := find_books(code,auth,title);
                          count := 1;
                          cases len books {
                            0    => message(6,WARN,.. );
                            1    => fm_view(hd books,"");
```

```
                        TRUE => message(6,NOTE,...);
                    };
                };
    command " NEXT " => { if books = <> then
                            message(6,WARN,...)
                        else {
                            fm_view(hd books,...),
                            count := count+1;
                            books := tl books;
                            message(6,NOTE,...);
                        };
                };
    command " BACK " => message(6,NOTE,"") => fields;
    command " QUIT " => exit;
};
```

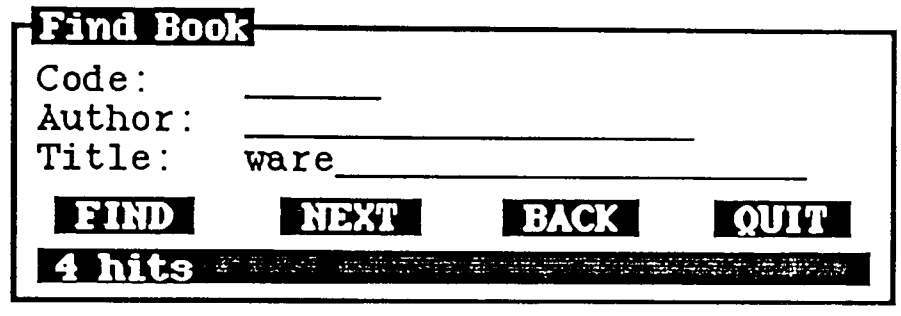


FIGURE 8.3 A dialogue box for finding books.

The dialogue box defines a number of fields and commands. Each field consists of a field name, a field variable, the length of the field, and a value which depicts an empty field. In the fields part, the symbol => depicts a transfer of control to commands. Each command consists of a command name and a corresponding action. In the commands part, the symbol => depicts the association of an action with a command, and also the transfer of control to fields.

The effect of the above dialogue box is that it first allows the user to supply a book code. If the user does so control will be transferred to the commands. The user can then select a command and execute it. If no book code is given, the user will be asked for an author name and a book title. If either of these, or both, is given then control will be transferred to the commands, otherwise the whole process will be repeated, i.e. the user will be asked for a book code etc. When in the command section, the user can FIND books, look at the NEXT book if more than one book is found, go BACK to the fields part, or QUIT the dialogue box. The number of fields and commands is only limited by the physical size of the screen.

As illustrated in chapter 7, complex states allow the abstraction of an entire STI Clusters allow the abstraction of STDs along other dimensions; a recurring pattern in STD can be abstracted and supported by a cluster-defined notation. For example, the dialogue bo above corresponds to a specific pattern in the STDs of a library system (see appendix C.2.)

8.6 DISCUSSION

A higher order abstraction technique based on user-defined syntax rules which, i contrast to normal abstraction techniques, allows one to treat non-elementary components c a language such as statements and expressions as objects, can be a highly useful tool i software development. It allows one to manipulate the very things a language is compose of, and to extend the base language in directions which cannot, in general, be predicted i advance.

An additional level of abstraction of this kind has two advantages. First, it allows th formulation and encapsulation of concepts which have been developed by others, but whic cannot be conveniently captured by conventional means. Second, important abstractions ca be developed and integrated into the base language, thereby extending its capabilities toward the needs of its users. One can also envisage the use of this form of abstraction for deriving from the base language, languages which are geared towards specialised applications. Th potentials of all this for prototyping is obviously tremendous.

Some of the meta abstraction techniques described in this chapter are also available i certain programming languages. Clusters, for example, share with LISP the idea of direc manipulation of expressions in an unevaluated form. The concept of a programmabl syntax-driven module interface, however, is unique to clusters and in not supported by an other language.

Chapter 9 CASE STUDIES

Good judgement comes from experience.
Experience comes from bad judgement.
- J Horning

In addition to numerous small published programs [Jones80, Bjorner79, Bjorner82] EPROS has also been applied to three relatively large problems. These problems are o increasing size and complexity and are described in the following sections. The first two ar based on published VDM specifications and address functionality only. The last problem wa entirely specified and developed by the author and considers functionality as well as use interface.

9.1 ABSTRACT MAPPINGS

This study was based on the VDM specification of Fielding [Fielding80] for binary anc B^+ trees [Comer79]. These specifications have been formally verified, refined, and implemented in Pascal by the original author. The study involved converting the specifications to a suitable form for EPROS, compile them, and evaluate the resulting prototypes. The conversion task was straightforward; only three lines in the entire specification of the B^+ tree had to be changed. No changes were required for the specification of the binary tree.

The results were quite interesting. The specification of the B^+ tree contained an erroi which had been overlooked by Fielding, even in the formal proofs. This error was discovered by the EPROL compiler. However, no other errors were found during the evaluations. The study is summarised in figure 9.1.

specification	size (lines)	man days effort	errors in the specification
binary tree	93	0.5	0
B+ tree	118	1	1

FIGURE 9.1 Abstract mappings case study summary.

9.2 A VERSION CONTROL PROGRAM

This study was based on the VDM specification of Cottam [Cottam84] for a system version control program (SVCP). An SVCP is a program which records the interdependency relations of the documents for a software system and is used to keep track of different versions of the system (especially its source code). Like the previous study, the conversion of the specification to the EPROL notation was straightforward and no changes were required.

The study confirmed the correctness of the specification and no errors were detected during the compilation and evaluation sessions. The study is summarised in figure 9.2.

specification	size (lines)	man days effort	errors in the specification
SVCP	168	1.5	0

FIGURE 9.2 SVCP case study summary.

9.3 A LIBRARY SYSTEM

The last case study was based on developing a computerised system to automate the daily functions of a conventional library. This study is useful for two reasons. Firstly, the system to be discussed corresponds to a real world problem of considerable size. It puts into practice the techniques, described in earlier chapters, in the context of a realistic project. Secondly, it gives an idea of the effort involved in our development method. In particular, it provides some rough productivity measures for evolutionary prototyping.

requirements

The requirements for the system were derived from the procedures for the Open University library. This is a manual library of moderate size. The procedures cover six volumes of written text and are relatively complicated. After an initial study of the procedures, a simplified set of requirements were derived.

The major simplifications of the requirements were:

- The system will only deal with books and no other forms of publication.
- A keyboard will be used as the data entry device instead of a light pen.

- The system will be single user to avoid concurrency problems.
- Apart from the usual reports, the system will not generate any statistical data on the activities of the library.
- A year will be assumed to consist of 12 months each 30 days long.

The actual requirements will not be presented in full here. The following is an informal overview of some of the more important requirements:

- The library must provide functions for dealing with reader registration/deregistration and purchasing, issuing, discharging, reserving, recalling and renewing books.
- Each reader must register with the library. Each registered reader is allocated an id. number.
- For each registered reader the system must record the following: reader name etc., joining date, expected leaving date and the books he or she has on loan.
- Each book is allocated a code number for the purpose of identification.
- For each book the system must record the author, title, volume number etc.
- A reader may borrow up to 40 books.
- The loan period for a book is 14 days. After this period the reader must renew the book or return it to the library.
- If a reader does not return or renew a book after 14 days it will be recalled by the library.
- If a recalled book is not returned after 30 days it will be recalled again.
- A book may be recalled up to 4 times.
- If a reader does not return a book after 200 days it will be assumed lost.
- A reader whose entire loan is assumed lost is deregistered immediately and may no borrow again from the library.
- A reader is deregistered when he or she leaves, provided the loan has been returned to the library.
- A reader who has left, but has not returned his or her loan, will remain registered until he or she does so, or until the entire loan is assumed lost. In the mean time, the reader will not be allowed to borrow any more books.
- A book already on loan to a reader may be reserved by any other reader provided the reserving reader is within the loan limit.
- There is no limit on the number of readers who may reserve the same book.
- There is no limit on the number of books a reader may reserve.
- When a reserved book becomes available it will be offered to the first person who has reserved it. The reader is given 14 days to collect the book otherwise it will be offered to the next reader in the reservation queue.

- The library records shall be updated on a daily basis.
- The system should produce reports of new readers, new books, deregistered readers, lost books, released books, additions to the stock etc.

The library system was developed in four cycles. Each cycle is briefly described below. For a more detailed description of the system see [Hekmatpour87].

cycle 1

During the first cycle a formal specification of the functional requirements was produced. The specification was then compiled and the resulting prototype was evaluated. A few iterations then followed during which a number of errors and shortcomings in the specification were detected and corrected. When the specification reached an acceptable level, it was formally verified. However, no further errors were detected. The end product of this cycle is the formal specification given in appendix D.1.

cycle 2

The user interface to the system was specified as a hierarchy of state transition diagrams, the latest version of which is given in appendix D.2. The user interface was realised in a crude form and subjected to evaluation. A few dialogue errors were detected and subsequently corrected. The evaluations led to a number of improvements in the specification of the dialogue.

cycle 3

The user interface was improved in a number of respects. The simple command line based interface was gradually replaced by a menu driven interface. Also, the information held about readers and books was extended and these were designed as electronic forms. Following some evaluation of the generated prototypes, a number of dialogue boxes were designed which made the user interface more convenient and suggestive.

cycle 4

The last cycle involved making the system more concrete. For example, the in-core

databases were replaced by external files, and many abstractions were realised by more concrete constructs, e.g. a number of sets were realised as liked lists. As refinement progressed, the specification part shrank and the implementation part grew steadily. Eventually, the specification part vanished completely and the system reached a fully concrete form. A number of modifications were also made to the design of the system. For instance, two dialogue boxes which contained a number of fields for data entry and a number of associated commands were generalised and converted into a cluster. During the evaluations of the system, only one error was detected which corresponded to the original specification. All other errors were refinement errors. The final code of the system is given in appendix D.3. It consists of 1 cluster module, 6 dialogue modules, 2 form modules, and 22 function modules. The development cycles are summarised in figure 9.3.

	size (lines)	man days effort	errors in this cycle	errors in previous cycles
cycle 1	262	5	6	0
cycle 2	405	4	4	0
cycle 3	758	4	5	0
cycle 4	1253	5	5	1

FIGURE 9.3 Summary of the development cycles.

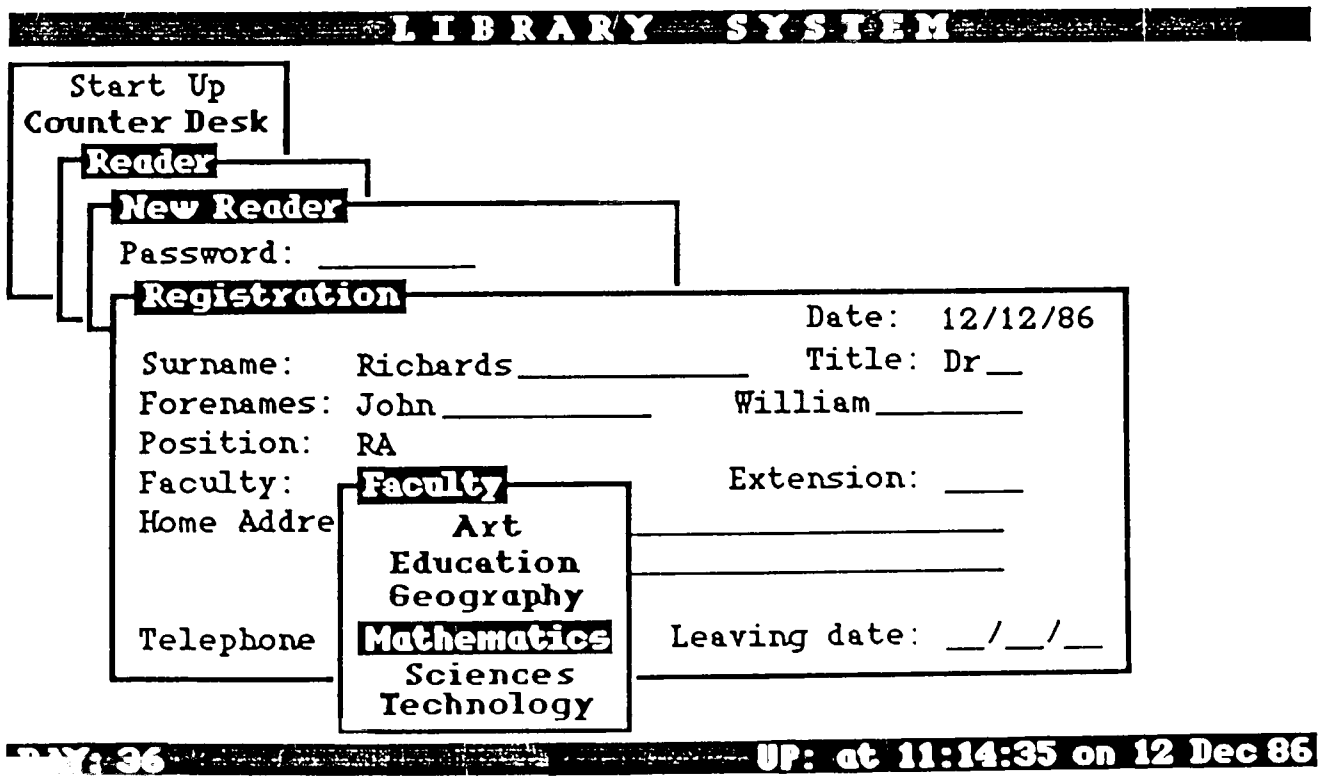


FIGURE 9.4 Registering a reader in the library system.

The final system has a clean design, is well modularised and relatively efficient. The user interface is hierarchical and quite friendly in operation. Figure 9.4 shows a snap shot of a dialogue with the system when filling a reader registration form.

concluding remarks

The experience gained from the development of the library system may be summarised as follows:

- The development process was smooth and no major problems were experienced.
- The results of evaluation sessions were often surprising, exposing errors which were least expected. For example, despite the simplicity of the operation REM_READ (see appendix D.1), two errors were detected in its specification.
- Although the functional specification formulated in the first cycle was formally verified, nevertheless, an error got through and was only detected in the last cycle.
- Numerous syntactic and semantic errors were detected automatically by the EPROL compiler.
- All other errors were discovered by evaluation of the prototypes. These errors often surfaced very quickly after a few minutes of use.
- The use of prototypes was most helpful in deciding the appearance of the user interface. Again, the results were surprising here; what appeared good on paper was usually different on the screen. For example, the layout of one of the dialogue boxes was changed a number of times before deciding on its final form.
- The use of prototyping in a disciplined way resulted in a clean and modular design. Much of this cleanness is due to the first two cycles.
- The decision to formulate functionality before specifying the user interface was helpful and worked very well.
- The overall development was very fast and productive. Use of prototypes allowed us to compare the merits of many design decisions in a short period of time and adopt the ones which were most satisfactory.

It should, however, be noted that this exercise was not carried out in as realistic a condition as we would have desired. For example, there were no real customers involved and the development team consisted of one programmer only. Feedback on the use of the prototypes was obtained from colleagues who were willing to play the role of a customer. These simplifications, of course, could not be avoided since the resources required for

simulating a 'real world' experiment were not available and could not be provided.

Despite these limitations, the library system was a good exercise in evaluating the potentials of EPROS for evolutionary prototyping and at least substantiates our claims of the appropriateness of the presented methods for prototyping non-trivial systems. We suspect that had we followed a conventional method, the development would have required much more effort and would have resulted in a system of lower quality.

Chapter 10 CONCLUSIONS

We presented a broad and comprehensive view of rapid prototyping and its role in software development. A system was described which is novel in a number of respects and which provides direct support for and integrates a number of prototyping techniques. It was demonstrated how the system could be beneficial in prototyping both the functional and the dialogue aspects of a software system, and how these prototypes could evolve within the system towards the final product. We also illustrated, through a case study, how the evolutionary prototyping approach could be made practical and productive, using this system.

In this chapter, we shall look at the work carried out by other researchers in this area and compare it to the research described here. The chapter ends with a discussion of potential research avenues for the future.

10.1 RELATED WORK

Rapid prototyping is a relatively new topic in software engineering. Because of its newness, not much work has been done in this area and research has only been intensified in the past two or three years. The existing literature on the subject, although small, shows a wide range of ideas and attempts, of which, the following are related to the work presented here.

executable specification systems

The idea of constructing a system which automatically generates a working prototype from a formal specification is not new and has already been pursued by other researchers. A number of such systems were described in chapter 3. Most of these systems, however, are either too elementary [Darlington83, Belkhouche85, Goguen84, Farkas82, Lee85] or geared towards specific applications [Urban85, Zave86, McGowan85]. A common fault of current systems is that they lack the concept of data abstraction. The Ina Jo system of Kemmerer [Kemmerer85] is a notable exception; this system, however, is currently based on symbolic

execution and is unable to produce realistic prototypes.

There have been two previous attempts to produce executable specification systems for subsets of VDM. Henderson [Henderson85] describes a system called ME-TOO which is based on a functional language and has most of its features borrowed from VDM and MIRANDA [Turner84]. LDM [Farkas82] is another system based on a subset of VDM, but simpler in some respects. Again, both these systems have no facilities for specifying abstract data types and instead rely on pure functions.

EPROS is an improvement over the above systems in three respects. First, its functional specification notation is more comprehensive. In fact, it is the largest VDM-based system to the author's knowledge. Second, it offers a number of additional useful features which are non-existent in other systems (e.g. abstract data type protection and polymorphic types.) Third, unlike similar systems which are interpreted, it provides a compiler as well as an interpreter for executing specifications. The use of a compiler is rather crucial for large specifications.

An interesting use of formal specifications has been reported by McMullin [McMullin83]. He describes a compiler-based system called DAISTS [Gannon81] which combines the algebraic specification of an abstract data type with its implementation. DAISTS uses the former as a test driver for the latter. Exactly the same principle is supported by EPROS.

As noted in chapter 5, the relationship between a specification and its refinement can be documented by a retrieve function. This function can play the role of a test oracle [Weyuker82] to ensure that the behaviour of an implementation matches that of its specification. DAISTS requires the developer to define a function which checks the equality of objects in an abstract data type, for no purpose other than producing the oracle. EPROS avoids this overhead since the equality operator is fully generalised and works for any object.

A number of other researchers have constructed and/or used abstract programming languages as executable specification notations. The languages used for this purpose include PROLOG [Lee85, Kowalski85, Tavendale85], SETL [Levin83], and MIRANDA [Turner84]. A common drawback of these languages is their very restricted and primitive I/O facilities.

Amongst these notations, MIRANDA is probably the most powerful. It is a purely functional language and has some advanced features such as polymorphic types and currying. Disregarding the syntactical differences, however, nearly the entire notation of MIRANDA can be viewed as a subset of the functional specification notation of EPROL.

Semi-formal notations have also been used for the automatic generation of prototypes. These include data flow diagrams [Olson85, Docker86], Petri nets [Bruno85], and requirement statement languages [Bell77]. Unfortunately, because of their choice of notation, these systems lack the facilities expected of a general purpose prototyping tool and are more useful for simulating very specific aspects of an application (e.g. flow of data in a control system) than prototyping.

application generators

Application generators are systems with a non-procedural front-end which enable users to generate an application after a short sequence of interaction with the system [Horowitz84, Read81, Lucker86]. A number of such systems were described in chapter 3. The most significant advantage of application generators is their high productivity. Also, the user needs to know little about the system. This makes them exceptionally suitable for inexperienced end-users who are interested in producing their own applications. Obviously, these system can also be valuable prototyping tools.

The serious limitation of these systems, however, is their very restricted scope. Their use is often confined to database manipulation and report generation in applications such as stock control and accounting [Martin82, Ramamoorthy84]. A few application generators, notably QBE/OBE [Zloff81], have gone a step further by integrating knowledge about general data processing, word processing and graphics into the system.

The essential differences between an application generator and EPROS is their scope and intended audience. In contrast to the former, the latter is for experienced software engineers and has a much wider application scope.

program transformation systems

The basic idea behind this approach is to initially produce an abstract and concise program which is generally inefficient. This program is then refined using transformation rules which are either supplied interactively by the user or suggested automatically by the system [Knuth74, Loveman77, Darlington81b, Bastani84]. The purpose of the transformations is to either refine or optimise the program.

Obviously, program transformations, when attainable, can be very valuable in prototyping. However, research into program transformation has been slow and has had very limited success so far. One reason for this is the difficulties associated with discovering correct and useful transformations. Another reason is the question of detecting parts of a program to which transformations should be applied [Wegbreit76]. The current body of knowledge on program transformation is quite limited [Barstow85] and much work remains to be done before it can be of serious utility in large software projects.

An interesting and more practical application of program transformation has been implemented in the DRACO system [Neighbours84]. This system relies on creating large transformation databases for specific application domains. Neighbours reports that he has successfully constructed a number of large applications using DRACO [Neighbours81]. A similar approach is described in [Rice81]. Although the use of domain specific transformation is attractive, two research issues remain to be explored. One is related to the potential difficulties of analysing a domain in great depth; another is related to the growth and size of domain languages which have to be mastered in order to use the system [Horowitz84].

EPROS does not utilise any program transformation techniques. We do suspect, however, that should program transformation become sufficiently practical in the future, the functional specification notation of EPROL will be a suitable candidate for applying these techniques.

program refinement systems

Cheatham [Cheatham79b] describes a program development system (PDS) where the

levels of refinement of a program are formally managed by the system. The system uses a database to maintain multiple representations of a program module and is based on an extensible language. In PDS, a module can be modified, either by manual editing or by applying rewrite rules, to generate another version of the module. The commands used for this can be saved in the database and later on, in the event of module modification, used to replay the derivation sequence. PDS can obviously be useful as a support tool for writing reusable modules and hence for prototyping. Unfortunately, however, the use of rewrite rules requires the indepth understanding of a module structure, since these operate like Lisp macros and must build program fragments piece by piece.

EPROS does not provide any automatic support for managing the levels of refinement of a program, and requires the programmer to do this manually. The addition of a suitable database, however, could provide such a facility. Currently, this is being planned as an extension to the system.

formal program development environments

Latham [Latham85] describes a formal program development environment based on the OBJ algebraic specification language [Goguen84] and a subset of Pascal, called abstract Pascal. Programs in this system are first specified in OBJ and then manually implemented in abstract Pascal. The system also provides support for the partial proof of correctness of programs with respect to their specifications.

Many systems of this kind have been developed in the past; see for example [Deutsch69, German75, Tamir80, Shaw81]. Typically these systems consist of a verification condition generator which automatically generates assertions about programs using some heuristics, and an interactive theorem prover which assists the user in proving the correctness (or otherwise) of the generated assertions. Some systems also provide a compiler and a run-time environment for the implementation language used in the system.

EPROS is similar to these system in its use of a formal specification notation only. It differs from these systems in the way it utilises formal specifications in software development. The former uses a formal specification as a basis for generating rapid

prototypes. The latter use it as a basis for semi-mechanical program proofs. Although EPROS takes program proofs also into account, it takes a purely manual approach to this. This is a conscious decision and there are three reasons for it. First, mechanical theorem proving technology has not reached a stage to guarantee the possibility of proving any theorem in first order predicate calculus (especially non-trivial ones) and is not likely to do so. This, in turn, puts some serious limitations to the utility of the approach for practical applications. Second, program proofs at the code level are more time-consuming and less productive than at the design level (see [Jones80] for an excellent discussion of this issue) and, indeed, many reports on cost estimation of software errors strongly support this view [Boehm81]. Third, one of the most useful side-effects of a formal proof is that the person attempting the proof learns a great deal about the specification and the ways in which it may be simplified or improved. With automatic approaches this advantage is practically lost.

user interface management systems

A user interface management system (UIMS) is a software tool which frees application programs from low-level I/O details [Green85, Ramamoorthy86]. Regardless of its actual form, it provides an abstract notation for describing a user interface. In a way a UIMS is similar to a database management system [Buxton83]. The latter manages the communication between a program and its data, hiding away details about the internal organisation of data. The former plays a similar role between a program and its I/O events.

A number of user interface management systems have been previously constructed [Edmonds84, Jacob83, Hays85, Wasserman85, Bos83, Hartson84], mostly in the area of computer graphics [Hanau80, Olsen 83, Olsen84, Buxton83, Kasik82, Hagen85, Myers86]. These systems invariably achieve abstraction by restricting their application domains [McLean86, Hutchins86].

EPROS is similar to some of these system in its use of state transition diagrams. It has two important features which are not possessed by most other systems. First, it supports the view of dialogue refinement and encapsulation [Green85]. Second, it is application domain-independent. This is in complete contrast to RAPID/USE [Wasserman86], for

example, which uses a single level of state transition diagrams and is geared towards database manipulation.

The UIMS component of EPROS is based on the screen management library (scr) and is different from the above systems in the following way. Rather than providing a pre-defined and fixed notation, EPROS relies on a set of library routines and I/O primitives for dialogue design. This set has been intentionally kept small to simplify its use. Higher level notations are constructed by the programmer using clusters. In this way the programmer can bend the UIMS in many different ways and come up with notations that match the application at hand more naturally. No such facility exists in other systems.

executable dialogue abstractions

Certain dialogue concepts are so commonly used in interactive system that it pays to have abstractions that support them directly. Examples are electronic forms [Tsichritzis79], pop-up menus [Brown82] and dialogue boxes, and were extensively described in earlier chapters.

Currently, there are office automation systems that support the user definition of some of these concepts, for example forms, in a rudimentary way [Tsichritzis80, Fikes80, Bass85]. Other researchers have come up with notations that are abstract but are either too application specific [Rowe83] or not implemented [Gehani82b, Lafuente78].

Compared to these, the dialogue abstractions of EPROL have a number of advantages: they are general purpose, fully executable, abstract, and user definable/extensible. Some of these abstractions, however, have benefited from the existing unimplemented notations (e.g. Gehani's notation for forms [Gehani82b].)

10.2 WHAT IS NEW ABOUT THIS RESEARCH

The EPROS environment is a contribution to research on software prototyping, software development environments and language theory. This work is important in a number of ways:

- It is an attempt to produce a software development environment for evolutionary

prototyping where a working system is available during *all* the phases of development, starting at the highest level of specification and finishing with concrete code. To the author's knowledge there are no similar systems with such comprehensive capabilities.

- It is an attempt to produce a software development environment for developers who wish to use the executable specification approach to prototyping and yet allow the human-computer interface to be prototyped as well. Most current systems, for example [Henderson86, Urban85, Kemmerer85, Goguen79], cater only for functional aspects of prototyping. Others [Wasserman86] support both but are application dependent. Since a significant part of many systems consists of the user interface, we feel that any environment for prototyping, be it evolutionary or throw-it-away, should also support user interface development.

- It is an attempt to remove the notational barriers between successive stages of software development and provide support for the entire life cycle. The result of this is smoother communication between various experts of the development team and avoidance of the problem of having to cope with widely differing notations for different phases. In this, it is only similar to the work reported by Bauer [Bauer78, Bauer81]. This work does involve a wide spectrum language. However, it is not fully executable and also ignores the human-computer interface.

- Unlike other systems which take a simplistic view of dialogue design and restrict themselves to simple string oriented dialogues [Wassermann85, Jacob83, Edmonds84, Hanau80, Hartson84], the EPROS environment supports the prototyping of modern interaction concepts such as windows, pop-up menus and forms, which are becoming increasingly popular and contributing to more user-friendly interfaces.

- As a by-product of the need to produce an environment for evolutionary prototyping, we have devised an executable wide spectrum language which is capable of improving the efficiency of the formal development process. For example, the normal process of formal software development using VDM consists of a series of steps which are rigorously verified during which errors are discovered and removed. Past experience [Cottam84] suggests that even the simplest errors can involve large amounts of paper rework and can be excessively time consuming. We have found that the automatic syntactic and semantic checkings built into our processors expose these errors very quickly without costly mathematical verification.

- As a result of our attempt to simplify the task of prototyping software systems, we have devised a new meta abstraction technique which facilitates the encapsulation o

non-trivial concepts. This technique is a departure from the usual methods of procedural abstraction and considerably simplifies the task of developing reusable software modules. Its utility, however, is not restricted to prototyping; it can also be used profitably in software design.

10.3 FUTURE RESEARCH DIRECTIONS

Naturally, this thesis does not claim to have found all the answers. Indeed, the nature of some of the unresolved issues implies that many more years of research is needed before comprehensive conclusions can be drawn, and before we can claim to have the ideal means for prototyping. There are a number of areas where further research could prove beneficial; these are discussed below.

One potential research area, which we may consider as a direct extension of this work, would involve the construction of a prototyping environment which progressively produces more efficient prototypes. We achieved this goal, to some extent, by gradually moving along a notation spectrum, from the abstract to the detailed. Better results can be obtained by also improving our translation techniques. This may involve the direct translation of our notations into machine code (as opposed to Lisp in our system) and the use of sophisticated optimisation techniques. Although there is currently a wealth of knowledge available on advanced compilation techniques, the problem of applying these to prototyping environments, such as ours, still remains outstanding. An obvious payoff of such research would be more efficient environments for evolutionary prototyping where finished products can compete in terms of efficiency with those produced using conventional methods.

A second area of research would concentrate on inventing improved notations and techniques for prototyping. Of particular interest would be a unified notation and framework for function and user interface prototyping. Although we made some progress towards this in this thesis, there still remains a wealth of questions that need to be explored. For example, can methods be invented where functionality and dialogue can be derived from one another? Can systems be built which extract information from previous developments to guide future developments? Can AI techniques be of any benefit in these respects?

There are also a number of existing notations which could form a suitable basis for

prototyping new concepts. One such concept, which we did not consider in this thesis, is concurrency. Recent developments in computer science have lead to some powerful notations for expressing concurrency [Milner80, Manna81, Inmos84, Hoare85, Zave86]. Some work has also been carried out on the application of these notations to prototyping certain aspects of software systems (for example, interaction [Alexander86].) More research is needed in order to fully exploit the power of these notations and, particularly, to investigate how these notations may be integrated with others, such as those described in this thesis.

A third area of research would focus on devising more effective and accessible front-ends to prototyping environments. There are a number of existing technologies which could contribute towards this. For example, syntax directed editors [Teitelbaum81] could speed up development and reduce errors, and bit map display-based workstations [Smith82a, Webster83] could provide a suitable basis for the direct use of graphical notations [Reader85, Reiss86] for prototyping.

The last area of research would concentrate on applying the outcome of the research into prototyping to a realistic number of real-life projects. Such research will be highly empirical with the aim of generating valuable feedback which would be used in the development of a coherent prototyping methodology. This research could have a number of useful outcomes. First, it may provide data on the impact of the project size and nature of application on the effectiveness of the prototyping approach. Second, it may increase our understanding about how a prototype system should be designed. Third, the results could provide sensible answers to some of the management problems that the prototyping approach generates [Canning81, Keus82].

In parallel to these, effort should be put into recording and preparing the findings of research on prototyping for use by software practitioners. Most software developers hesitate to use prototyping because they know little about it and there is little material available in a suitable form to guide them. Continuous formulation of new techniques and tools for prototyping into a set of prototyping procedures could remedy this problem to some extent.

REFERENCES

[Aaram84] J. Aaram, "The BOP Prototyping Concept," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 179-187 (1984).

[Ackford67] R. L. Ackford, "Management Misinformation Systems," *Management Science*, Vol. 14(4) pp. 456-461 (1967).

[Aggleton86] P. Aggleton, "Towards Effective Prototyping," *Computing* - Software Notebook, 3 parts: pp. 35-36, 52-53, 39 (Oct.-Nov. 1986).

[Alavi84] M. Alavi, "An Assessment of Prototyping Approach to Information System Development," *Communications of the ACM*, Vol. 27(6) pp. 556-563 (1984).

[Alexander86] H. Alexander, *Formally-Based Tools and Techniques for Human-Computer Dialogues*, PhD Thesis, Dept. of Computing, Stirling University (1986).

[Alter80] S. A. Alter, *Decision Support Systems - current practice and continuing challenges*, Addison-Wesley, Reading MA (1980).

[Ardis86] M. A. Ardis, "Comparison of Algebraic and State-Machine Specification Methods," *ACM SIGSOFT Software Engineering Notes*, Vol. 11(4) pp. 54-56 (1986).

[Backus78] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM*, Vol. 21(8) pp. 613-641 (1978).

[Baldwin82] R. R. Baldwin, "Reportage on Spring 1982 IEEE COMPCON Conference," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(2) pp. 13-20 (1982).

[Bally77] L. Bally, J. Brittan, and K. H. Wagner, "A Prototype Approach to Information System Design and Development," *Information & Management*, Vol. 1 pp. 21-26 (1977).

[Balzer82] R. Balzer, N. M. Goldman, and D. S. Wile, "Operational Specification as the Basis of Rapid Prototyping," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 3-16 (1982).

[Balzer83] R. Balzer, T. E. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," *IEEE Computer*, Vol. 16(11) pp. 39-45 (1983).

[Barstow85] D. Barstow, "On Convergence Towards a Database of Program Transformations," *ACM Trans. Programming Languages and Systems*, Vol. 7(1) pp. 1-9 (1985).

[Basili75] V. R. Basili and D. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Trans. Software Engineering*, Vol. 1(4) pp. 462-471 (1975).

[Bass85] L. J. Bass, "An Approach to User Specification of Interactive Display Interfaces," *IEEE Trans. Software Engineering*, Vol. 11(8) pp. 686-698 (1985).

[Bastani84] F. A. Bastani, "Performance Improvement of Abstractions Through Context Dependent Transformations," *IEEE Trans. Software Engineering*, Vol. 10(1) pp. 100-116 (1984).

[Bastani85] F. A. Bastani, "Experience with a Feedback Version Development Methodology," *IEEE Trans. Software Engineering*, Vol. 11(8) pp. 718-723 (1985).

[Bauer78] F. L. Bauer et. al., "Towards a Wide Spectrum Language to Support Program Specification and Program Development," *ACM SIGPLAN Notices*, Vol. 13(12) pp.

15-24 (1978).

[Bauer81] F. L. Bauer et. al., "Programming in a Wide Spectrum Language: A Collection of Examples," *Science of Computer Programming*, Vol. 1 pp. 73-114 (1981).

[Beichter84] F. W. Beichter, O. Herzog, and H. Petzsch, "SLAN-4 - A Software Specification and Design Language," *IEEE Trans. Software ENgineering*, Vol. 10(2) pp. 155-162 (1984).

[Belady80] L. A. Belady and B. Leavenworth, "Program Modifiability," in *Software Engineering*, H. Freeman and P. M. Lewis (eds.), Academic Press, New York (1980).

[Belkouche85] B. Belkhouche, "Compilation of Specification Languages as a Basis for Rapid and Efficient Prototyping," in *Proc. 3rd Int. Workshop Software Specification & Design*, London, pp. 16-19 (1985).

[Bell77] T. E. Bell, D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Trans. Software Engineering*, Vol. 3(1) pp. 49-60 (1977).

[Bell79] Bell Lab., *UNIX Programmer's Manual*, 7th Edition (1979).

[Benbasat84] I. Benbasat and Y. Wand, "A Structured Approach to Designing Human-Computer Dialogues," *Int. Journal of Man-Machine Studies*, Vol. 21 pp. 105-126 (1984).

[Berrisford79] T. Berrisford and J. Wetherbe, "Heuristic Development: A Redesign of Systems Design," *MIS Quarterly*, Vol. 3(1) pp. 11-19 (1979).

[Berzins85] V. Berzins and M. Gray, "Analysis and Design in MSG.84: Formalizing Functional Specifications," *IEEE Trans. Software Engineering*, Vol. 11(8) pp. 657-670 (1985).

[Bird84] R. S. Bird, "The Promotion and Accumulation Strategies in Transformational Programming," *ACM Trans. Prog. Langs. and Systems*, Vol. 6(4) pp. 487-504 (1984).

[Bjorner78] D. Bjorner and C. B. Jones (eds.), *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science, Vol. 61, Springer Verlag, Berlin (1978).

[Bjorner82] D. Bjorner and C. B. Jones, *Formal Specification and Software Development*, Prentice-Hall, London (1982).

[Blesser82] T. Blesser and J. D. Foley, "Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces," in *Proc. Conf. Human Factors in Compter Systems*, Gaithersburg, Maryland, pp. 309-314 (1982).

[Bloomfield86] R. E. Bloomfield and P. K. D. Froome, "The Application of Formal Methods to the Assessment of High Integrity Software," *IEEE Trans. Software Engineering*, Vol. 12(9), pp. 988-993 (1986).

[Blum82a] B. I. Blum and R. C. Houghton, Jr., "Rapid Prototyping of Information Management Systems," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 35-38 (1982).

[Blum82b] B. I. Blum, "The Life Cycle - A Debate Over Alternate Models," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(4) pp. 18-20 (1982).

[Blum83] B. I. Blum, "Still More About Rapid Prototyping," *ACM SIGSOFT Software Engineering Notes*, Vol. 8(3) pp. 9-11 (1983).

[Blum86] B. I. Blum, "Iterative Development of Information Systems: A Case Study," *Software Practice and Exprerience*, Vol. 16(6) pp. 503-515 (1986).

[Boehm74] B. W. Boehm, "Some Steps Towards Formal and Automated Aids to Software Requirements Analysis and Design," *IFIP 74*, North-Holland, pp. 192-197 (1974).

[Boehm81] B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, New Jersey (1981).

[Boehm83] B. W. Boehm and T. A. Standish, "Software Technology in the 1990's: Using an Evolutionary Paradigm," *IEEE Computer*, Vol. 16(1) pp. 30-37 (Nov. 1983).

[Boehm84] B. W. Boehm , T. E. Gray, and T. Seewaldt, "Prototyping Versus Specifying: A Multiproject Experiment," *IEEE Trans. Software Engineering*, Vol. 10(3) pp. 290-303 (1984).

[Bonet84] R. Bonet and A. Kung, "Structuring into Subsystems: the experience of a prototyping approach," *ACM SIGSOFT Software Engineering Notes*, Vol. 9(5) pp. 23-27 (1984).

[Bos78] J. Bos, "Definition and Use of Higher-Level Graphics Input Tools," *Computer Graphics*, Vol. 12(3) pp. 38-42 (1978).

[Bos83] J. Bos, M. J. Plasmeijer, and P. H. Hartel, "Input-Output Tools: A Language Facility for Interactive and Real-Time Systems," *IEEE Trans. Software Engineering*, Vol. 9(3) pp. 247-259 (1983).

[Bosman81] A. Bosman and H. G. Sol, "Evolutionary Development of Information Systems," in *TC8 Working Conference on Evolutionary Information Systems*, Part 1 pp. (Sep. 1981).

[Botting85] R. J. Botting, "On Prototyping vs. Mockups vs. Breadboards," *ACM SIGSOFT Software Engineering Notes*, Vol. 10(1) p. 18 (Jan. 1985).

[Bourne83] S. R. Bourne, *The Unix System*, Addison-Wesley, London (1983).

[Boyle84] J. M. Boyle and N. M. Muralidharan, "Program Reusability Through Program Transformation," *IEEE Trans. Software Engineering*, Vol 10(5), pp. 574-588 (1984)

[Britwistle73] G. M. Birtwistle et. al., *Simula Begin*, Petrocelli, New York (1973).

[Brittan80] J. N. G. Brittan, "Design for a Changing Environment," *The Computer Journal*, Vol. 23(1) pp. 13-19 (1980).

[Brooks75] F. Brooks, *The Mythical Man-Month*, Addison-Wesely, Reading MA (1975).

[Brown82] J. W. Brown, "Controlling The Complexity of Menu Networks," *Communications of the ACM*, Vol. 25(7) pp. 412-418 (1982).

[Browne86] D. P Browne, "The Formal Specification of Adaptive User Interfaces Using Command Language Grammar," in *Proc. CHI'86*, Boston MA, pp. 256-260 (1986).

[Bruno85] G. Bruno and G. Marchetto, "Rapid Prototyping of Control Systems Using High Level Petri Nets," in *Proc. 8th Int. Conf. Software Engineering*, pp. 230-235 (1985).

[Budde84] R. Budde and K. Sylla, "From Application Domain Modelling to Target System," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 31-48 (1984).

[Burns86] A. Burns and J. A. Kirkham, "The Construction of Information Management System Prototypes in Ada," *Software Practice and Experience*, Vol. 16(4), pp. 341-350 (1986).

[Burstall80] R. M. Burstall, D. MacQueen, and D. Sanella, "HOPE: an Experimental Applicative Language," in *Conf. Rec. 1980 Lisp Conf.*, Stanford Univ., pp. 136-143 (1980).

[Burstall81] R. M. Burstall and J. A. Goguen, "An Informal Introduction to Specifications Using CLEAR," in *The Correctness problem in Computer Science*, J. Moore (ed.), Academic Press, New York (1981).

[Buxton83] W. Buxton, M. R. Lamb, D. Sherman, and K. C. Smith, "Toward a Comprehensive User Interface Management System," *Computer Graphics*, Vol. 17(3) pp. 35-42 (1983).

[Canning81] R. G. Canning, "Developing Systems by Prototyping," *EDP Analyzer*, Vol. 19(9) pp. 1-14 (1981).

[Carey82] T. Carey, "User Differences in Interface Design," *IEEE Computer*, Vol. 15(3), pp. 14-20 (1982).

[Casey82] B. E. Casey and B. Dasarathy, "Modelling and Validating the Man-Machine Interface," *Software-Practice and Experience*, Vol. 12 pp. 557-569 (1982).

[Cheatham79a] T. E. Cheatham, G. H. Holloway and J. A. Townley, "Symbolic Evaluation and Analysis of Programs," *IEEE Trans. Software Engineering*, Vol. 5(4) pp. 402-417 (1979).

[Cheatham79b] T. E. Cheatham, J. A. Townley and G. H. Holloway, "A System for Program Refinement," in *Proc. 4th. Int. Conf. Software Engineering*, pp. 53-62 (1979).

[Cheatham84] T. E. Cheatham, "Reusability Through Program Transformations," *IEEE Trans. Software Engineering*, Vol. 10(5) pp. 589-594 (1984).

[Cheng84] T. T. Cheng, E. D. Lock, and N. S. Prywes, "Use of Very High Level Languages and Program Generation by Management Professionals," *IEEE Trans. Software Engineering*, Vol. 10(5) pp. 552-563 (1984).

[Chi85] U. I. Chi, "Formal Specification of User Interface: A Comparison and Evaluation of Four Axiomatic Approaches," *IEEE Trans. Software Engineering*, Vol. 11(8) pp. 671-685 (1985).

[Christensen84] N. Christensen and K. Kreplin, "Prototyping of User-Iterfaces," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 58-67 (1984).

[Clark81] I. A. Clark, "Software Simulation as a Tool For Usable Product Design," *IBM System Journal*, Vol. 20(3) pp. 272-293 (1981).

[Clark84] F. Clark, P. Drake, M. Kapp and P. Wong, "User Acceptance of Information Technology Through Prototyping," in *Proc. Interact'84 Conf.*, London, pp. 274-279 (1984)

[Claybrook82] B. G. Claybrook, "A Specification Method for Specifying Data and Procedural Abstractions," *IEEE Trans. Software Engineering*, Vol. 8(5) pp. 449-459 (1982).

[Clocksin84] W. E. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 2nd edition (1984).

[Cohen82] D. Cohen, W. Swartout, and R. Balzer, "Using Symbolic Execution to Characterize Behaviour," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 25-32 (1982).

[Comer79] D. Comer, "The Ubiquitous B-Tree," *Computing Surveys* Vol. 11(2) pp. 121-137 (1979).

[Conway63] M. E. Conway, "Design of a Separable Transition Diagram Compiler," *Communications of the ACM*, Vol. 6(7) pp. 396-408 (1963).

[Cook86] S. Cook, "Modelling Generic User-Interface with Functional Programs," in *People and*

Computers: Designing for Usability , M. D. Harrison and A. I. Monk (eds.), Cambridge Univ. Press, London, pp. 368-385 (1986).

[Cottam84] I. Cottam, "Rigorous Development of a Version Control Program," *IEEE Trans. Software Engineering*, Vol. 10(2) pp. 143-154 (1984).

[Cristian84] F. Cristian, "Correct and Robust Programs," *IEEE Trans. Software Engineering*, Vol. 10(2) pp. 163-174 (1984).

[Dagwell83] R. Dagwell and R. Weber, "System Designers' User Models: A Comparative Study and Methodological Critique," *Communications of the ACM*, Vol. 26(11) pp. 987-997 (1983).

[Dahl72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London (1972).

[Dannenberg82] R. B. Dannenberg and G. W. Ernst, "Formal Program Verification Using Symbolic Execution," *IEEE Trans. Software Engineering*, Vol. 10(1) pp. 43-52 (1982).

[Darlington76] J. Darlington and R. M. Burstall, "A System which Automatically Improves Programs," *Acta Informatica*, Vol. 6 pp. 41-60 (1976).

[Darlington81a] J. Darlington, "An Experimental Program Transformation and Synthesis System," *Artificial Intelligence*, Vol. 16 pp. 1-46 (1981).

[Darlington81b] J. Darlington, "The Structured Description of Algorithm Derivation," in *Algorithmic Languages*, J. W. De Bakker and J. C. Van Vliet (eds.), North-Holland, Amsterdam, pp. 221-250 (1981).

[Darlington82] J. Darlington, P. Henderson, and D. A. Turner (eds.), *Functional Programming and its Applications - an advanced course*, Cambridge Univ. Press, Cambridge (1982).

[Darlington83] J. Darlington, "Validation Techniques for Software Specifications," in *Microcomputers: Developments in Industry, Business and Education*, C. J. van Spronsen (ed.), North-Holland, pp. 91-97 (1983).

[Davis77] C. G. Davis and C. R. Vick, "The Software Development System," *IEEE Trans. Software Engineering*, Vol. 3(1) pp. 69-84 (1977).

[Davis79] A. M. Davis, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications," in *Proc. Conf. Specification of Reliable Software*, pp. 15-35 (1979).

[Davis83] R. Davis, "Task Analysis and User Errors: A Methodology for Assessing Interactions," *Int. Journal of Man-Machine Studies*, Vol. 19 pp. 561-574 (1983).

[Dearnley81] P. A. Dearnley and P. Mayhew, "Experiments in Generating System Prototypes," in *Proc. First European Workshop on Inf. Systems Teaching*, Aix-en-Provence (1981).

[Dearnley83] P. A. Dearnley and P. J. Mayhew, "In Favour of System Prototypes and their Integration into the System Development Cycle," *The Computer Journal*, Vol. 26(1) pp. 36-42 (1983).

[Dearnley84] P. A. Dearnley and P. J. Mayhew, "On the Use of Software Development Tools in the Construction of Data Processing System Prototypes," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 68-79 (1984).

[Denert77] E. Denert, "Specification and Design of Dialogue Systems with State Diagrams," pp. 417-424 in *International Computing Symposium 1977*, D. Ribbens (ed.), North-Holland

(1977).

[Deutsch69] L. P. Deutsch, *An Interactive Program Verifier*, PhD Thesis, University of California, Berkeley (1969).

[Dixon85] F. J. Dixon, "Simplifying Screen Specifications - the 'Full Screen Manager' Interface and 'Screen Form' Generating Routines," *The Computer Journal* Vol. 28(2) pp. 117-127 (1985).

[Docker86] T. W. G. Docker anf G. Tate, "Executable data Flow Diagrams," in *Software Engineering 86*, D. Barnes and P. Brown (eds.), Peter Peregrinus, Exeter, pp. 352-370 (1986).

[Dodd80] W. P. Dodd, "Prototype Programs," *IEEE Computer*, Vol. 13(2) p. 80 (Feb. 1980).

[Draper85] S. W. Draper and D. A. Norman, "Software Engineering for User Interfaces," *IEEE Trans. Software Engineering*, Vol. 11(3) pp. 252-258 (1985).

[Drosten84] K. Drosten, "Towards Executable Specifications Using Conditional Axioms," in *STACS 84*, Lecture Notes in Computer Science Vol. 166, Springer-Verlag, pp. 85-96 (1984).

[Dyer80] M. Dyer, "The Management of Software Engineering Part IV: Software Development Practices," *IBM System Journal*, No. 4 pp. 458-459 (1980).

[Earl78] M. J. Earl, "Prototype Systems for Accounting, Information and Control," *Accounting, Organisation and Society*, Vol 3(2) pp. 161-170 (1978).

[Edmonds81] E. A. Edmonds, "Adaptive Man-Computer Interfaces," in *Computing Skills and the User Interface*, eds. M. J. Coombs and J. L. Alty, Academic Press, pp. 389-426 (1981).

[Edmonds82] E. A. Edmonds, "The Man-Computer Interface: a note on concepts and design," *Int. Journal of Man-Machine Studies*, Vol. 16 pp. 231-236 (1982).

[Edmonds84] E. A. Edmonds and S. Guest, "The SYNICS2 Interface Manager," in *Proc. Interac 84, 1st IFIP Conf. Human Computer Interaction*, pp. 53-56 (1984).

[Farkas82] Z. Farkas, P. Szeredi and E. Santane-Toth, "LDM - A Program Specification Support System," in *Proc. Logic Programming Workshop*, Marcei-France, pp. 123-128 (1982).

[Feather82a] M. S. Feather, "Mappings for Rapid Prototyping," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 17-24 (1982).

[Feather82b] M. S. Feather, "Program Specification Applied to a Text Formatter," *IEEE Trans. Software Engineering*, Vol. 8(5) pp. 490-498 (1982).

[Feyock77] S. Feyock, "Transition Diagram Based CAI/HELP Systems," *Int. Journal of Man-Machine Studies*, Vol. 9 pp. 399-413 (1977).

[Fielding 80] E. Fielding, *The Specification of Abstract Mappings and their Implementation as B+ trees*, Msc Thesis, PRG, Oxfor University (1980).

[Fikes80] R. Fikes, "Odyssey: A Knowledge Based Assistant," *Xerox Research Centre*, Palo Alto CA (1980).

[Floyd84] C. Floyd, "A Systematic Look at Prototyping," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 1-18 (1984).

[Foley82] J. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading MA (1982).

[Ford85] R. Ford and K. Miller, "Abstract Data Type Development and Implementation," *IEEE Trans. Software Engineering*, Vol. **11**(10) pp. 1033-1037 (1985).

[Fox82] J. M. Fox, *Software and its Development*, Prentice-Hall, New Jersey (1982).

[Furtado85] A. L. Furtado and T. S. E. Maibaum, "An Informal Approach to Formal (Algebraic) Specifications," *The Computer Journal*, Vol. **28**(1) pp. 59-67 (1985).

[Gaines81] B. R. Gaines, "The Technology of Interaction - dialogue programming rules," *Int. Journal of Man-Machine Studies*, Vol. **14** pp. 133-150 (1981).

[Gannon81] J. Gannon, P. McMullin, and R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing," *ACM Trans. Programming Languages and Systems*, Vol. **3**(3) pp. 221-223 (1981).

[Gehani82a] N. H. Gehani, "A Study in Prototyping," *ACM SIGSOFT Software Engineering Notes*, Vol. **7**(5) pp. 71-75 (1982).

[Gehani82b] N. H. Gehani, "The Potential of Forms in Office Automation," *IEEE Trans. Communications*, Vol. **30**(1) pp. 120-125 (1982).

[Gehani83] N. H. Gehani, "High Level Form Definition in Office Information Systems," *The Computer Journal*, Vol. **26**(1), pp. 52-59 (1983).

[German75] S. M. German and B. Wegbreit, "A Synthesizer of Inductive Assertions," *IEEE Trans. Software Engineering*, Vol. **1**(1), pp. 68-75 (1975).

[Gilb81] T. Gilb, "Evolutionary Development," *ACM SIGSOFT Software Engineering Notes*, Vol. **6**(2) p. 17 (1981).

[Gilb85] T. Gilb, "Evolutionary Delivery versus the Waterfall Model," *ACM SIGSOFT Software Engineering Notes*, Vol. **10**(3) pp. 49-62 (1985).

[Gill82] H. Gill, R. Lindvall, O. Rosin, E. Sandewall, H. Sorensen, and O. Wigertz, "Experience from Computer Supported Prototyping for Information Flow in Hospitals," *ACM SIGSOFT Software Engineering Notes*, Vol. **7**(5) pp. 67-70 (1982).

[Gittins84] D. T. Gittins, R. L. Winder, and H. E. Bez, "An Icon-Driven End-User Interface to UNIX," *Int. Journal of Man-Machine Studies*, Vol. **21** pp. 451-461 (1984).

[Gladden82] G. R. Gladden, "Stop the Life-Cycle, I Want to Get off," *ACM SIGSOFT Software Engineering Notes*, Vol. **7**(2) pp. 35-39 (1982).

[Glass81] R. L. Glass and R. A. Noiseux, *Software Maintenance Guidebook*, Prentice-Hall, New Jersey (1981).

[Glass82] R. L. Glass, "Recommended: A minimum Standard Software Toolset," *ACM SIGSOFT Software Engineering Notes*, Vol. **7**(4) pp. 3-13 (1982).

[Goguen79] J. A. Goguen and J. J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing formal Algebraic Program Specifications," in *Proc. Specification of Reliable Software*, pp. 170-189 (1979).

[Goguen84] J. A. Goguen, "Parameterised Programming," *IEEE Trans. Software Engineering*, Vol. 10(5) pp. 528-543 (1984).

[Gomaa81] H. Gomaa and D. B. H. Scott, "Prototyping as a Tool in the Specification of User Requirements," in *Proc. 5th Int. Conf. Software Engineering*, pp. 333-342 (1981).

[Gomaa83] H. Gomaa, "The Impact of Rapid Prototyping on Specifying User Requirements," *ACM SIGSOFT Software Engineering Notes*, Vol. 8(2) pp. 17-28 (1983).

[Good84] D. Good, J. A. Whiteside, D. R. Wixon, and S. J. Jones, "Building a User-Derived Interface," *Communications of the ACM*, Vol. 27(10) pp. 1032-1043 (1984).

[Goodwin81] J. W. Goodwin, "Why Programming Environments Need Dynamic Data Types," *IEEE Trans. Software Engineering*, Vol. 7(5) pp. 451-457 (1981).

[Gordon79] M. J. Gordon, A. J. Milner and C. P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science, Vol 78, Springer-Verlag, Berlin (1979).

[Gould83] J. D. Gould , J. Conti, and T. Hovanyecz, "Composing Letters with a Simulated Listening Typewriter," *Communications of the ACM*, Vol. 26(4) pp. 295-308 (1983).

[Gray85] D. Gray and A. Kilgour, "Guide: A UNIX-Based Dialogue Design System," in *People and Computers: Designing the Interface*, P. Johnson and S. Cook (eds.), Cambridge Univ. Press, Cambridge, pp. 148-160 (1985).

[Green81] M. Green, "A Methodology for the Specification of Graphics User Interface," *Computer Graphics*, Vol. 15(3) pp. 99-108 (1981).

[Green85] M. Green, "Design Notations and User Interface Management Systems," in *UIMS*, G. E. Pfaff (ed.), Springer-Verlag, Berlin (1985).

[Gregory84] S. T. Gregory, "On Prototypes vs. Mockups," *ACM SIGSOFT Software Engineering Notes*, Vol. 9(5) p. 13 (1984).

[Griswold71] R. E. Griswold, J. F. Poage and I.P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, New Jersey (1971).

[Groner79] C. Groner, M. D. Hopwood, N. A. Palley and W. Sibley, "Requirements Analysis in Clinical Research Information Processing - A Case Study," *IEEE Computer*, Vol. 12(9) pp. 100-108 (1979).

[Guest82] S. P. Guest, "The Use of Software Tools for Dialogue Design," *Int. Journal of Man-Machine Studies*, Vol. 16 pp. 263-285 (1982).

[Guttag77] J. V. Guttag, "Abstract Data Types and the Development of Data Structures," *Communications of the ACM*, Vol. 20(6) pp. 396-404 (1977).

[Guttag78] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, Vol. 10 pp. 27-52 (1978).

[Hagen80] P. J. W. Ten Hagen, "A Conceptual Basis for Graphical Input and Output Interaction," in *Methodology of Interaction*, R. A. Guedi et. al. (eds.), North-Holland, Amsterdam, pp. 239-246 (1980).

[Hagen85] P. J. W. Ten Hagen and J. Dresken, "Parallel Input and Feedback in Dialogue Cells," in *UIMS*, G. E. Pfaff (ed.), Springer-Verlag, Berlin, pp. 109-124 (1985).

[Hall86] P. A. V. Hall, "Reusable and Reconfigurable Software Using C," in *Software Engineering 86*, D. Barnes and P. Brown (eds.), Peter Peregrinus, pp. 164-174 (1986).

[Hanau80] P. R. Hanau and D. R. Lenorovitz, "Prototyping and Simulation Tools for User/Computer Dialogue Design," *Computer Graphics*, Vol. 14(2) pp. 271-278 (1980).

[Hansal76] A. Hansal, "A Formal Definition of a Relational Database System," *IBM UKSC 0080 Report* (1976).

[Hartson84] H. R. Hartson, D. H. Hix and R. W. Ehrich, "A Human-Computer Dialogue Management System," in *Proc. INTERACT '84*, North-Holland, Amsterdam, pp. 379-384 (1984).

[Hawgood82] J. Hawgood (ed.), *Evolutionary Information Systems*, North-Holland, Amsterdam (1982).

[Hayes81] P. J. Hayes, E. Ball, and R. Reddy, "Breaking the Man-Machine Communication Barrier," *IEEE Computer*, Vol. 14(3) pp. 19-30 (1981).

[Hayes85] P. J. Hayes, P. A. Szekely and R. A. Lerner, "Design Alternatives for User Interface Management Systems Based on Experience with COUSIN," in *Proc. CHI '85*, San Fransisco, pp. 169-176 (1985).

[Heitmeyer82] C. Heitmeyer, C. Landwehr, and M. Cornwell, "The Use of Quick Prototypes in the Secure Military Message Systems Project," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 85-87 (1982).

[Hekmatpour86] S. Hekmatpour, "EPROS: A Software Engineer's Manual," *Tech. Report No. 86/2*, Mathematics Faculty, Open University (1986).

[Hekmatpour87] S. Hekmatpour and D. C. Ince, *Software Prototyping, Formal Methods and VDM*, Addison-Wesley (to appear).

[Henderson80] P. Henderson, *Functional Programming - Application and Implementation*, Prentice-Hall, London (1980).

[Henderson82] J. C. Henderson and R. S. Ingraham, "Prototyping for DSS: A Critical Appraisal," in *Decision Support Systems*, E. A. Stohr (ed.), North-Holland, pp. 79-96 (1982).

[Henderson84] P. Henderson, "me-too - A Language for Software Specification and Model Building - preliminary report," *Tech. Report FPN-9*, Computing Dept., Stirling University (1984).

[Henderson85] P. Henderson and C. Minkowitz, "The me-too Method of Software Design," *Tech. Report FPN-10*, Computing Dept., Stirling University (1985).

[Henderson86a] P. Henderson, "Functional Programming, Formal Specification, and Rapid Prototyping," *IEEE Trans. Software Engineering*, Vol. 12(2) pp. 241-250 (1986).

[Henderson86b] P. Henderson and C. Minkowitz, "Software Design Using Executable Formal Specifications - a Consideration of two Approaches," *Tech. Report FPN-12*, Computing Dept., Stirling University (1986)

[Hewett86] T. T. Hewett, "The Role of Interactive Evaluation in Designing Systems for Usability," in *People and Computers: Designing for Usability*, M. D. Harrison and A. I. Monk (eds.), Cambridge Univ. Press, London, pp. 196-214 (1986).

[Hoare73] C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," *Acta Informatica*, Vol. 2 pp. 335-355 (1973).

[Hoare 85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, London (1985).

[Hoeve84] F. A. van Hoeve and R. Engmann, "The TUBA-Project: A Set of Tools for Application Development and Prototyping," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 202-213 (1984).

[Hopgood80] F. R. Hopgood and D. A. Duce, "A Production Approach to Interactive Graphic Program Design," in *Methodology of Interaction*, R. A. Guedj et. al. (eds.), pp. 247-263 (1980).

[Hooper82] J. W. Hooper and P. Hsia, "Scenario-Based Prototyping of Requirements Identification," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 88-93 (1982).

[Horowitz84] E. Horowitz and J. B. Munson, "An Expansive View of Reusable Software," *IEEE Trans. Software Engineering*, Vol. 10(5) pp. 477-487 (1984).

[Hutchins86] E. L. Hutchins, J. D. Hollan and D. A. Norman, "Direct Manipulation Interfaces," in *User Centred System Design*, D. A. Norman and S. W. Draper (eds.), Lawrence Erlbaum, New Jersey (1986).

[Iivari84] J. Iivari, "Prototyping in the Context of Information Systems Design," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 261-277 (1984).

[Inmos84] Inmos Ltd., *OCCAM Programming Manual*, Prentice-Hall, London (1984).

[Jackson85] M. I. Jackson, "Developing Ada Programs Using the Vienna Development Method (VDM)," *Software Practice and Experience*, Vol. 15(3) pp. 305-318 (1985).

[Jacob83] R. J. K. Jacob, "Using Formal Specifications in The Design of a Human-Computer Interface," *Communications of the ACM*, Vol. 26(4) pp. 259-264 (1983).

[James80] E. B. James, "The User Interface," *The Computer Journal*, Vol. 23(1) pp. 25-28 (1980).

[Johnson68] W. L. Johnson, "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques," *Communications of the ACM*, Vol. 11(12) pp. 805-813 (1968).

[Johnson75] S. C. Johnson, "Yacc: Yet Another Compiler Compiler," *Comp. Sci. Tech. Report No. 32*, Bell Lab, Murray Hill NJ (1975).

[Jones77] C. B. Jones, "Program Specification and Formal Development," in *International Computing Symposium 1977*, D. Ribbens (ed.), North-Holland, pp. 537-553 (1977).

[Jones80a] C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, London (1980).

[Jones80b] C. B. Jones, "The Role of Formal Specification in Software Development," *Life Cycle Management: Infotech State of the Art Report*, Vol. 8(7) pp. 117-133 Infotech Ltd. (1980).

[Jones86] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall, London (1986).

[Kant81] E. Kant and D. R. Barstow, "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis," *IEEE Trans. Software Engineering*, Vol. 7(5) pp. 458-471 (1981).

[Kasik82] D. J. Kasik, "A User Interface Management System," *SIGGRAPH '82*, p. 99 (1982).

[Keen81] P. G. W. Keen, "Information Systems and Organisational Change," *Communications of the ACM*, Vol. 24(1), pp. 24-33 (1981).

[Kemmerer85] R. A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," *IEEE Trans. Software Engineering*, Vol. 11(1) pp. 32-43 (1985).

[Kennedy75] K. Kennedy and J. Schwartz, "An Introduction to The Set Theoretical Language SETL," *Comp. & Maths with Applications*, Vol. 1 pp. 97-119 (1975).

[Kenneth81] C. R. C. Kenneth, "Screen Updating and Cursor Movement Optimization: A Library Package," *Tech. Report*, Dept. of Electrical Eng. and Comp. Sci., Univ. of California,

Berkeley (1981).

[Kernighan78a] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, New Jersey (1978).

[Kernighan78b] B. W. Kernighan and W. Plauger, *The Elements of Programming Style*, McGraw-Hill (1978).

[Kernighan84] B. W. Kernighan, "The UNIX System and Software Reusability," *IEEE Trans. Software Engineering*, Vol. 10(5) pp. 513-518 (1984).

[Keus82] H. E. Keus, "Prototyping: A More Reasonable Approach to System Development," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 94-95 (1982).

[Kieras83] D. Kieras and P. Polson, "A Generalized Transition Network Representation for Interactive Systems," in *Proc. CHI'83*, pp. 103-106 (1983).

[Knuth 74] D. E. Knuth, "Structured Programming with goto Statements," *Computing Surveys*, Vol. 6(4) pp. 261-301 (1974).

[Kowalski79] R. Kowalski, "Algorithm = Logic + Control," *Communications of the ACM*, Vol. 22(7) pp. 424-436 (1979).

[Kowalski85] R. Kowalski, "The Relation Between Logic Programming and Logic Specification," *Int. Journal of Man-Machine Studies*, Vol. 22(4) pp. 365-394 (1985).

[Kraushaar85] J. M. Kraushaar and L. E. Shirland, "A Prototyping Method for Applications Development by End Users and Information Systems Specialists," *MIS Quarterly*, Vol. 9(2), pp. 189-197 (1985).

[Kruchten84] P. Kruchten and E. Schonberg, "The Ada/Ed System: A Large-scale Experiment in Software Prototyping Using SETL," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 398-415 (1984).

[Kruesi83] E. Kruesi, "The Human Engineering Task Area," *IEEE Computer*, Vol. 16(1), pp. 86-93 (Nov. 1983).

[Lafuente78] J. M. Lafuente and D. Gries, "Language Facilities for Programming User-Computer Dialogues," *IBM Journal of Research & Development*, Vol. 22(2) pp. 145-158 (1978).

[Lanergan84] R. G. Lanergan and C. A. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Trans. Software Engineering*, Vol. 10(5), pp. 498-501 (1984).

[Latham85] J. T. Latham, *Abstraction in Program Verification*, PhD Thesis, Dept. of Computation, UMIST (1985).

[Leavenworth74] B. M. Leavenworth and J. E. Sammet, "Overview of Non-Procedural Languages," *ACM SIGPLAN Notices*, Vol. 9(2) pp. 98-103 (1974).

[Lee85] S. Lee and S. Sluizer, "On Using Executable Specifications for High-Level Prototyping," in *Proc. 3rd Int. Workshop Software Specification & Design*, London, pp. 130-134 (1985).

[Lehman85] J. D. Lehman and N. Yavneh, "The Total Life Cycle Model," in *Proc. 3rd Int. Workshop Software Specification & Design*, London, pp. 135-137 (1985).

[Leibrandt84] U. Leibrandt and P. Schnupp, "An Evaluation of Prolog as a PrototypingSystem," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 424-433 (1984).

[Lenorovitz77] D. R. Lenorovitz and H. R. Ramsey, "A Dialogue Simulation Tool for Use in the Design of Interactive Computer Systems," in *Proc. 21st Annual Meeting of Human Computer Factors Society*, Santa Monica CA, pp. 95-99 (1977).

[Levene82] A. A. Levene and G. P. Mullery, "An Investigation of Requirements Specification Languages: Theory and Practice," *IEEE Computer*, Vol. 15(1) pp. 50-59 (1982).

[Levine80] J. Levine , "Why a Lisp-Based Command Language?," *SIGPLAN Notices*, Vol. 15(5) pp. 49-53 (1980).

[Levin83] D. Levin, "Programming in SETL Environment," in *Programming Languages and System Design*, J. Bormann (ed.), North-Holland, pp. 129-137 (1983).

[Lientz80] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*, Addison-Wesley, Reading MA (1980).

[Lientz83] B. P. Lientz, "Issues in Software Maintenance," *Computing Surveys*, Vol. 15(3) pp. 271-278 (1983).

[Liskov75] B. H. Liskov and S. N. Zilles, "Specification Techniques for Data Abstractions," *IEEE Trans. Software Engineering*, Vol. 1(1), pp. 7-19 (Mar. 1975).

[Litvintchouk84] S. D. Litvintchouk and A. S. Matsumoto, "Design of Ada Systems Using Reusable Components: An Approach Using Structured Algebraic Specification," *IEEE Trans. Software Engineering*, Vol. 10(5), pp. 544-551 (1984).

[Loveman77] D. B. Loveman, "Program Development by Source-to-Source Translation," *Journal of the ACM*, Vol. 24(1) pp. 121-145 (1977).

[Luker86] P. A. Luker and A. Burns, "Program Generators and Generation Software," *The Computer Journal*, Vol. 29(4) pp. 315-321 (1986).

[MacEwen82] G. H. MacEwen, "Specification Prototyping," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 112-119 (1982).

[MacLennan83] B. J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt Rinehart and Winston, New York (1983).

[Mallgren82] W. R. Mallgren, "Formal Specification of Graphic Data Types," *ACM Trans. Programming Languages and Systems*, Vol. 4(4) pp. 687-710 (1982).

[Manna81] Z. Manna and A. Pneuli, "Verification of Concurrent Programs: the Temporal Logic Framework," in *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore (eds.), Academic Press, London, pp. 215-273 (1981).

[Martin82] J. Martin, *Application Development Without Programmers*, Prentice-HAll, New Jersey (1982).

[Mason82] R. E. A. Mason and T. T. Carey, "ACT/1: A Tool for Information Systems," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 120-126 (1982).

[Mason83] R. E. A. Mason and T. T. Carey, "Prototyping Interactive Information Systems," *Communications of the ACM*, Vol. 26(5) pp. 347-354 (1983).

[Matsumoto84] A. S. Matsumoto, "Some Experience in Promoting Reusable Software Presentation in Higher Abstract Levels," *IEEE Trans. Software Engineering*, Vol. 10(5) pp. 502-512 (1984).

[Mayr84] H. C. Mayr, M. Bever, and P. C. Lockemann, "Prototyping Interactive Application

Systems," in *Approaches to Prototyping*, Springer-Verlag, pp. 105-121 (1984).

[McCracken82] D. D. McCracken and M. A. Jackson, "Life Cycle Concept Considered Harmful," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(2) pp. 29-32 (1982).

[McGowan85] C. L. McGowan, M. D. Feblowitz, and M. Chandrasekharan, "The Metafor Approach to Executable Specifications," in *Proc. 3rd Int. Workshop Software Specification & Design*, London, pp. 163-169 (1985).

[McLean76] E. R. McLean, "The Use of APL for Production Applications: The Concept of 'Throwaway Code'," in *APL 76: Conf. Proc., ACM* (1976).

[McMullin83] P. R. McMullin and J. D. Gannon, "Combining Testing with Formal Specifications: A Case Study," *IEEE Trans. Software Engineering*, Vol. 9(3) pp. 328-334 (1983).

[McNurlin81] B. C. McNurlin, "Developing Systems by Prototyping," *EDP Analyzer*, Vol. 19(10), pp. 1-12 (1981).

[Meandzija86] B. Meandzija, "A Formal Method for Composing a Network Command Language," *IEEE Trans. Software Engineering*, Vol. 12(8) pp. 860-865 (1986).

[Meijer79] E. Meijer, "Application Simulation," in *Proc. DESIGN'79 Symp.*, Monterey CA, pp. 410-420 (1979).

[Meurs77] J. Van Meurs and E. L. Cardozo, "Interfacing the User," *Software Practice and Experience*, Vol. 7(1) pp. 85-93 (1977).

[Meyer78] G. J. Meyer, *The Art of Software Testing*, John Wiley & Sons, New York (1978).

[Meyer82] B. Meyer, "Principles of Package Design," *Communications of the ACM*, Vol. 25(7) pp. 419-428 (1982).

[Mills85] J. A. Mills, "A Pragmatic View of The System Architect," *Communications of the ACM*, Vol. 28(7) pp. 708-717 (1985).

[Milner80] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science Vol. 92, Springer-Verlag, New York (1980).

[Minkowitz86] C. Minkowitz and P. Henderson, "A Formal Description of Object-Oriented Programming Using VDM," *Tech. Report FPN-13*, Computing Dept., Stirling University (1986).

[Mittermeir82a] R. T. Mittermeir, "HIBOL, A Language for Fast Prototyping in Data Processing Environments," *ACM SIGSOFT Software Engineering Notes* Vol. 7(5) pp. 133-140 (1982).

[Mittermeir82b] R. T. Mittermeir, "Semantic Nets for Modelling the Requirements of Evolvable Systems - an Example," in *Evolutionary Information Systems*, J. Hawgood (ed.), North-Holland, pp. 193-216 (1982).

[Moran81] T. P. Moran, "The Command Language Grammer: A representation for the user interface of interactive computer systems," *Int. Journal of Man-Machine Studies*, Vol. 15 pp. 3-50 (1981).

[Morgan84] C. Morgan and B. Sufrin, "Specification of the UNIX Filing System," *IEEE Trans. Software Engineering*, Vol. 10(2) pp. 128-142 (1984).

[Mumford78] E. Mumford, F. Land, and J. Hawgood, "A Participative Approach to The Design of Computer Systems," *Impact of Science on Society*, Vol. 28(3) pp. 235-253 (1978).

[Munson81] J. B. Munson, "Software Maintainability: A Practical Concern for Life-Cycle Costs," *IEEE Computer*, Vol. 14(2) pp. 103-109 (1981).

[Musser79] D. R. Musser, "Abstract Data Type Specification in the AFFIRM System," in *Proc. Specification of Reliable Software*, pp. 47-57 (1979).

[Myers86] B. A. Myers and W. Buxton, "Creating Highly Interactive and Graphical Interfaces by Demonstration," *SIGGRAPH*, Vol. 20(4) pp. 249-258 (1986).

[Naumann82] J. D. Naumann and A. M. Jenkins, "Prototyping: The New Paradigm for Systems Development," *MIS Quarterly*, pp. 29-44 (Sep. 1982).

[Naur63] P. Naur et. al., "Revised Report on the Algorithmic Language Algol 60," *Comm. of the ACM*, Vol. 6(1), (1963).

[Neighbours81] J. M. Neighbours, *Software Construction Using Components*, PhD Thesis, University of California, Irvine (1981).

[Neighbours84] J. M. Neighbours, "The Draco Approach to Constructing Software From Reusable Components," *IEEE Trans. Software Engineering*, Vol. 10(5), pp. 564-578 (1984).

[Norman83] D. A. Norman , "Design Rules Based on Analysis of Human Error," *Communications of the ACM*, Vol. 26(4) pp. 254-258 (1983).

[Nosek84] J. T. Nosek, "Organisation Design Choices to Facilitate Evolutionary Development of Prototype Information Systems," in *Approaches to Prototyping*, R. Budde et. a. (eds.), Springer-Verlag, pp. 341-355 (1984).

[Olsen83] D. R. Olsen, Jr., "Automatic Generation of Interactive Systems," *ACM Computer Graphics*, Vol. 17(1) pp. 53-57 (1983).

[Olsen84] D. R. Olsen, W. Buxton, R. Ehrich, D. J. Kasik, J. R. Rhyne and J. Silbert, "A Context for User Interface Management," *IEEE Computer Graphics and Applications*, Vol. 4 pp. 33-42 (1984).

[Olson85] C. Olson, W. Webb, and R. Wieland, "Code Generation from Data Flow Diagrams," in *Proc. 3rd Int. Workshop Software Specification & Design*, London, pp. 172-176 (1985).

[Pagan83] F. G. Pagan, "A Diagrammatic Notation for Abstract Syntax and Abstract Structured Objects," *IEEE Trans. Software Engineering*, Vol. 9(3) pp. 280-289 (1983).

[Parnas69] D. L. Parnas , "On The Use of Transition Diagrams in The Design of A User Interface for an Interactive Computer System," in *Proc. 24th Nat. ACM Conf.*, pp. 379- 385 (1969).

[Parnas72] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15(12) pp. 1053-1058 (1972).

[Parnas79] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Software Engineering*, Vol. 5(2) pp. 128-137 (1979).

[Parnas85] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The Modular Structure of Complex Systems," *IEEE Trans. Software Engineering*, Vol. 11(3) pp. 259-272 (1985).

[Parnas86] D. L. Parnas and P. C. Clements, "A Relational Design Process: How and Why to Fake it," *IEEE Trans. Software Engineering*, Vol. 12(2) pp. 251-257 (1986).

[Patton83] B. Patton, "Prototyping - A Nomenclature Problem," *ACM SIGSOFT Software Engineering Notes*, Vol. 8(2) pp. 14-16 (1983).

[Podger79] D. N. Podger, "High-Level Languages - A Basis for Participative Design," in *Design and Implementation of Computer-Based Information Systems*, E. Grochla (ed.), Sijthoff & Noordhoff (1979).

[Polster86] F. J. Polster, "Reuse of Software Through Generation of Partial Systems," *IEEE Trans. Software Engineering*, Vol. 12(3) pp. 402-416 (1986).

[Prywes83] N. S. Prywes and A. Pnueli, "Compilation of Nonprocedural Specifications into Computer Programs," *IEEE Trans. Software Engineering*, Vol. 9(3) pp. 267-279 (1983).

[Ramamoorthy84] C. V. Ramamoorthy, A. Prakash, W. Tsai, and Y. Usuda, "Software Engineering: Problems and Perspectives," *IEEE Computer*, Vol. 17(2) pp. 191-209 (1984).

[Ramamoorthy86] C. V. Ramamoorthy, V. Garg and A. Parkash, "Programming in the Large," *IEEE Trans. Software Engineering*, Vol 12(7) pp. 769-783 (1986).

[Read81] N. S. Read and D. L. Harmon, "Assuring MIS Success," *Datamation*, Vol. 27(2), pp. 109-120 (1981).

[Reader85] G. Reader, "A Survey of Current Graphical Programming Techniques," *IEEE Computer*, Vol. 18(8) pp. 11-25 (1985).

[Reisner81] P. Reisner, "Formal Grammar and Human Factors Design of an Interactive Graphics System," *IEEE Trans. Software Engineering*, Vol. 7(2) pp. 229-240 (1981).

[Reiss86] S. P. Reiss, "An Object-Oriented Framework for Graphical Programming," *ACM SIGPLAN Notices*, Vol. 21(10) pp. 49-57 (1986).

[Rice81] J. G. Rice, *Build Program Techniques: A Practical Approach for the Development of Automatic Software Generation Systems*, Wiley and Sons, New York (1981).

[Rich82] C. Rich and R. C. Waters, "The Disciplined Use of Simplifying Assumptions," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 150-154 (1982).

[Rich84] E. Rich, "Natural-Language Interfaces," *IEEE Computer*, Vol. 17(6) pp. 39-47 (1984).

[Riddle84] W. E. Riddle, "Advancing the State of the Art in Software System Prototyping," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 19-26 (1984).

[Ross77] D. T. Ross and K. E. Schoman, Jr., "Structured Analysis for Requirements Definition," *IEEE Trans. Software Engineering*, Vol. 3(1) pp. 6-15 (1977).

[Rowe81] L. A. Rowe, "Data Abstraction from a Programming Language Viewpoint," *SIGPLAN Notices*, Vol. 16(1) pp. 29-35 (1981).

[Rowe83] L. A. Rowe and K. Shoens, "Programming Language Constructs for Screen Definition," *IEEE Trans. Software Engineering*, Vol. 9(1) pp. 31-39 (1983).

[Rzevski84] G. Rzevski, "Prototypes versus Pilot Systems: Strategies for Evolutionary Information System Development," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 356-367 (1984).

[Sale85] A. E. Sale, "The Codasyl Proposal for a Screen Management Facility," *Computer Bulletin*, Vol. 1(1) pp. 24-25 (1985).

[Sandwall78] E. Sandewall, "Programming in an Interactive Environment: The Lisp Experience," *ACM Computing Surveys*, Vol. 10(1) pp. 35-71 (1978).

[Scott78] J. H. Scott, "The management Science Opportunity: A Systems Development Management Viewpoint," *MIS Quarterly*, Vol. 2(4) pp. 59-61 (1978).

[Shannon75] R. E. Shannon, *System Simulation - the Art and Science*, Prentice-Hall, Englewood Cliffs, New Jersey (1975).

[Shaw81] M. Shaw (ed.), *Alphard: Form and Content*, Springer-Verlag, New York (1981).

[Shaw83] M. Shaw, E. Boriston, M. Horowitz, T. Lane, D. Nichlos and R. Pausch, "Descartes: A Programming-Language Approach to Information Display Interfaces," *SIGPLAN Notices*, Vol 18(6), pp. 100-111 (1983).

[Shaw85] M. Shaw, "What Can We Specify? Issues in the Domains of Software Specifications," in *Proc. 3rd Int. Workshop Software Specification & Design*, London, pp. 214-215 (1985).

[Shaw86] M. Shaw, "An Input-Output Model for Interactive Systems," *Proc. CHI'86*, Boston MA, pp. 261-273 (1986).

[Shneiderman79] B. Shneiderman, "Human Factors Experiments in Designing Interactive Systems," *IEEE Computer*, Vol. 12(1) pp. 9-19 (Dec. 1979).

[Shneiderman82] B. Shneiderman, "Multiparty Grammars and Related Features for Defining Interactive Systems," *IEEE Trans. Systems, Man & Cybernetics*, Vol. 12(2) pp. 148-154 (1982).

[Shooman82] M. L. Shooman, *Software Engineering: Design, Reliability, and Management*, McGraw-Hill, New York (1982).

[Silverberg81] B. A. Silverberg, "An Overview of the SRI Hierarchical Development Methodology," in *Software Engineering Environments*, R.Bunke (ed.) , North-Holland, pp. 235-252 (1981).

[Silbert86] J. L. Silbert, W. D. Hurley and T. W. Bleser, "An Object-Oriented User Interface Management System," *SIGGRAPH*, Vol. 20(4) pp. 259-268 (1986).

[Smith76] H. R. Smith and C. Knuth, "A Computerised Approach to Systems Analysis: A Technique and its Application," in *Proc. 8th. Annual Conf. American Decision Sciences*, pp. 549-551 (1976).

[Smith82a] D. C. Smith, C. Irby, R. Kimball and B. Verplank, "Designing the Star User Interface," *BYTE*, pp. 242-282 (1982).

[Smith82b] D. A. Smith, *Rapid Software Prototyping*, PhD Thesis, University of California, Irvine (1982).

[Sol84] H. G. Sol, "Prototyping: A Methodological Assessment," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 368-382 (1984).

[Sommerville82] I. Sommerville, *Software Engineering*, Addison-Wesley, London (1982).

[Somogyi81] E. K. Somogyi, "Prototyping - A Method not to be Missed," *EDP Analyser*, Vol. 19(10) pp. (1981).

[Sprague80] R. H. Sprague, "A Framework for the Development of Decision Support Systems," *Management Inform. Systems Quarterly*, Vol. 4(4) pp. 1-26 (1980).

[Stoy82] J. Stoy, "Some Mathematical Aspects of Functional Programming," in *Functional Programming and its Applications*, J. Darlington et. al. (eds.), Cambridge Univ. Press, Cambridge, pp. 217-252 (1982).

[Sufrin82] B. Sufrin, "Formal Specification of a Display-Oriented Text Editor," *Science of*

Computer Programming, Vol. 1, pp. 157-202 (1982).

[Sufrin86] B. Sufrin, "Formal Methods and the Design of Effective User Interfaces," in *People and Computers: Designing for Usability*, M. D. Harrison and A. I. Monk (eds.), Cambridge Univ. Press, London, pp. 24-43 (1986).

[Sunshine82] C. A. Sunshine, D. H. Thompson, R. W. Erickson, S. L. Gerhart, and D. Schwabe, "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models," *IEEE Trans. Software Engineering*, Vol. 8(5) pp. 460-489 (1982).

[Sutton78] J. A. Sutton and R. H. Sprague, "A Study of Display Generation and Management in Interactive Business Applications," *IBM Research Rep. RJ2392* (1978).

[Swanson76] E. B. Swanson, "The Dimensions of Maintenance," in *Proc. 2nd Int. Conf. Software Engineering*, pp. 492-497 (1976).

[Swartout82] W. Swartout and R. Balzer, "On the Inevitable Interwining of Specification and Implementation," *Communications of the ACM*, Vol. 25(7) pp. 438-440 (1982).

[Taggart77] W. M. Taggart, Jr. and M. O. Tharp, "A survey of Information Requirements Analysis Techniques," *Computing Surveys*, Vol. 9(4) pp. 271-289 (1977).

[Tamir80] M. Tamir, "ADI: Automatic Derivation of Invariants," *IEEE Trans. Software Engineering*, Vol. 6(1) pp. 40-48 (1980).

[Tavendale85] R. D. Tavendale, "A Technique for Prototyping Directly from a Specification," in *Proc. 8th Int. Conf. Software Engineering*, pp. 224-229 (1985).

[Tavolato84] P. Tavolato and K. Vincena, "A Prototyping Methodology and its Tools," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 434-446 (1984).

[Teichroew77] D. Teichroew and E. A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Engineering*, Vol. 3(1) pp. 41-48 (1977).

[Teitelman79] W. Teitelman, "A Display Oriented Programmer'a Assistant," *Int. Journal of Man-Machine Studies*, Vol. 11 pp. 157-187 (1979).

[Teitelbaum81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax Directed Programming Environment," *Communications of the ACM*, Vol. 24(9) pp. 563-573 (1981).

[Tomeski75] E. A. Tomeski and H. Lazarus, *People-oriented Computer Systems*, Van Nostrand Reinhold, New York (1975).

[Tseng86] J. S. Tseng, B. Szymanski, Y. Shi and N. S. Prywes, "Real-Time Software Life Cycle with the Model System," *IEEE Trans. Software Engineering*, Vol. 12(2) pp. 358-373 (1986).

[Tsichritzis79] D. Tsichritzis, "A Form Manipulation System," *Tech. Report CSRG-101*, University of Toronto, pp. 53-71 (1979).

[Tsichritzis80] D. Tsichritzis, "OFS: An Integrated Form Management System," in *Proc. ACM Conf. Very Large Data Bases*, pp. 190-194 (1980).

[Tsichritzis82] D. Tsichritzis, "Form Management," *Communications of the ACM*, Vol. 25(7) pp. 453-478 (1982).

[Turner79] D. A. Turner, "A New Implementation Technique for Applicative Languages," *Software*

Practice and Experience, Vol 9(1), pp. 31-49 (1979).

[Turner84] D. A. Turner, "Functional Programs as Executable Specifications," *Phil. Trans. Royal Society of London*, Vol. 312 pp. 363-388 (1984).

[Turner85] D. A. Turner, *Miranda: A Non-Strict Functional Language with Polymorphic Types*, Lecture Notes in Computer Science Vol. 201, Springer-Verlag, Berlin (1985).

[Turoff82] M. Turoff, S. R. Hiltz, and E. B. Kerr, "Controversies in the Design of Computer-Mediated Communication Systems: A Delphi Study," in *Proc. Conf. Human Factors in Compter Systems*, Gaithersburg, Maryland, pp. 89-100 (1982).

[Urban82] J. E. Urban, "Software Development with Executable Functional Specifications," in *Proc. 6th. Int. Conf. Software Engineering*, Tokyo Japan, pp. 418-419 (1982).

[Urban85] S. D. Urban, J. E. Urban, and W. D. Dominick, "Utilizing an Executable Specification Language for an Information System," *IEEE Trans. Software Engineering*, Vol. 11(7) pp. 598-605 (1985).

[VanWyk82] C. J. Van Wyk, "A High-Level Language for Specifying Pictures," *ACM Trans. Graphics*, Vol. 1(2) pp. 163-182 (1982).

[Venken84] R. Venken and M. Bruynooghe, "Prolog as a Language for Prototyping of Information Systems," in *Approaches to Prototyping*, R. Budde et. al. (eds.), Springer-Verlag, pp. 447-458 (1984).

[Walter84] C. Walter, "Control Software Specification and Design: An Overview," *IEEE Computer*, Vol. 17(2) pp. 20-23 (Feb. 1984).

[Wang70] M. D. Wang, "The Rule of Syntactic Complexity as a Determinator of Comprehensibility," *Journal Verbal Learning and Verbal Behaviour*, Vol. 9 pp. 398-404 (1970).

[Wasserman79] A. I. Wasserman and S. K. Stinson, "A Specification Method for Interactive Information Systems," in *Proc. Conf. Specification of Reliable Software*, pp. 68-79 (1979).

[Wasserman82a] A. I. Wasserman and D. Shewmake, "Automating the Development and Evolution of User Dialogue in an Interactive Information System," in *Evolutionary Information Systems*, J. Hawgood (ed.), North-Holland, pp. 159-172 (1982).

[Wasserman82b] A. I. Wasserman and D. T. Shewmake, "Rapid Prototyping of Interactive Information Systems," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 171-180 (1982).

[Wasserman85] A. I. Wasserman, "Extending State Transition Diagrams for the Specification of Human-Computer Interaction," *IEEE Trans. Software Engineering*, Vol. 11(8) pp. 699-713 (1985).

[Wasserman86] A.I. Wasserman, P. A. Pircher, D. T. Shewmake and M. L. Kersten, "Developing Interactive Information Systems with the User Software Engineering Methodology," *IEEE Trans. Software Engineering*, Vol. 12(2) pp. 326-345 (1986).

[Waters79] S. J. Waters, "Towards Comprehensive Specifications," *The Computer Journal*, Vol. 22(3) pp. 195-199 (1979).

[Weber86] H. Weber and H. Ehrig, "Specification of Modular Systems," *IEEE Trans. Software Engineering*, Vol. 12(7) pp. 784-798 (1986).

[Webster83] R. Webster and M. Miner, "Apple Lisa," *Personal Computer World*, Vol. 6(7), pp. 146-159 (1983).

[Wegbreit76] B. Wegbreit, "Goal-Directed Program Transformation," *IEEE Trans. Software Engineering*, Vol. 2(2) pp. 69-80 (1976).

[Weiser82] M. Weiser, "Scale Models and Rapid Prototyping," *ACM SIGSOFT Software Engineering Notes*, Vol. 7(5) pp. 181-185 (1982).

[Weyuker82] E. J. Weyuker, "On Testing Non-Testable Programs," *Computer Journal*, Vol. 25(4), pp. 465-470 (1982).

[Wilensky84] R. Wilensky, *LISPcraft*, W.W. Norton & Company, New York (1984).

[Winston81] P. H. Winston and B. K. P. Horn, *Lisp*, Addison-Wesley, Reading MA (1981).

[Xerox81] The Xerox Learning Research Group, "The Smalltalk-80 System," *BYTE*, pp. 36-48 (Aug. 1981).

[Yao84] S. B. Yao, A. R. Hevner, A. Shi and D. Luo, "FORMANAGER: An Office Forms Management System," *ACM Trans. Office Information Systems*, Vol. 2(3), pp. 235-262 (1984).

[Young81] R. M. Young, "The Machine Inside the Machine: User's Models of Pocket Calculators," *Int. Journal of Man-Machine Studies*, Vol. 15, pp. 51-58 (1981).

[Zave81] P. Zave and R. T. Yeh, "Executable Requirements for Embedded Systems," in *Proc. 5th Int. Conf. Software Engineering*, pp. 295-304 (1981).

[Zave86] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and Its Environment," *IEEE Trans. Software Engineering*, Vol. 12(2) pp. 312-325 (1986).

[Zelkowitz79] M. V. Zelkowitz, A. C. Shaw and J. D. Gannon, *Principles of Software Engineering and Design*, Prentice-Hall (1979).

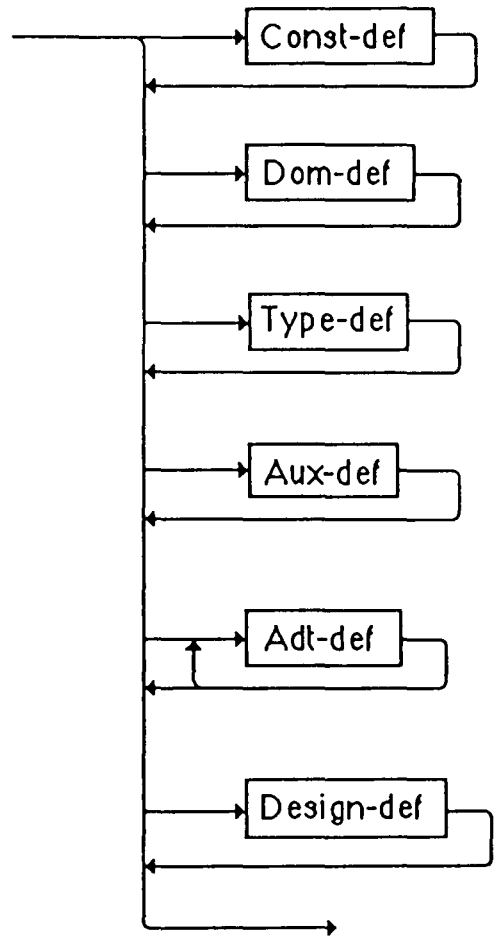
[Zelkowitz80] M. V. Zelkowitz, "A Case Study in Rapid Prototyping," *Software Practice and Experience* Vol. 10 pp. 1037-1042 (1980).

[Zelkowitz84] M. V. Zelkowitz, "A Taxonomy of Prototype Designs," *ACM SIGSOFT Software Eng. Notes*, Vol. 9(5) pp. 11-12 (1984).

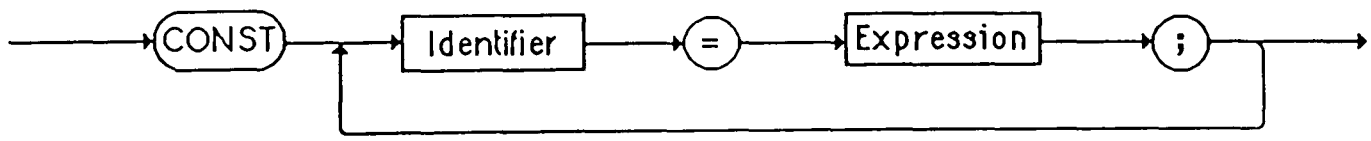
[Zloff81] M. M. Zloff, "QBE/OBE: A Language for Office and Business Automation," *IEEE Computer*, Vol 14(5) pp. 13-23 (1981).

Appendix A EPROL SYNTAX

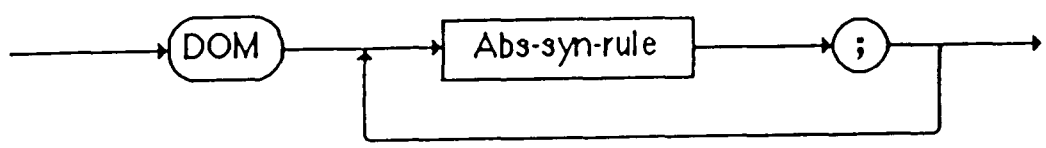
Prototype



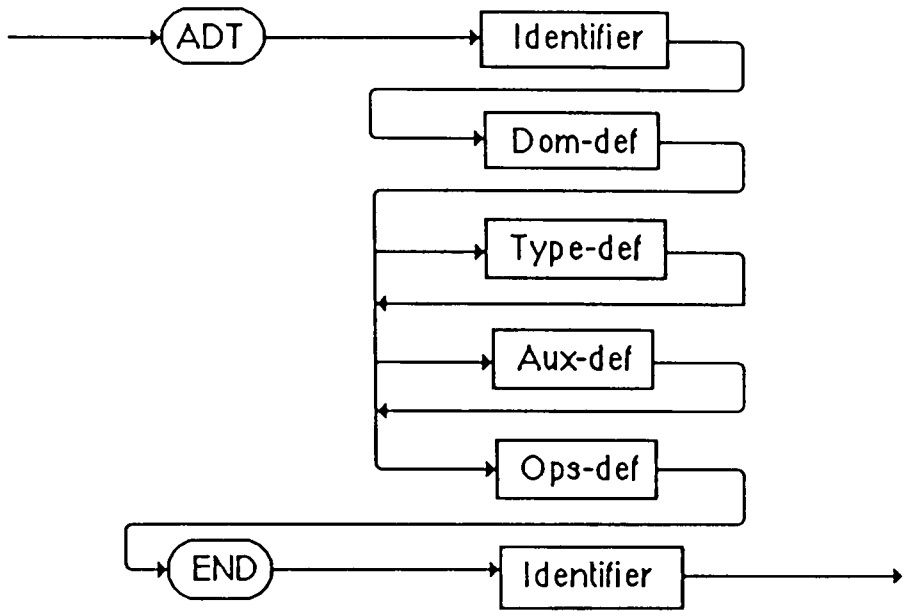
Const-def



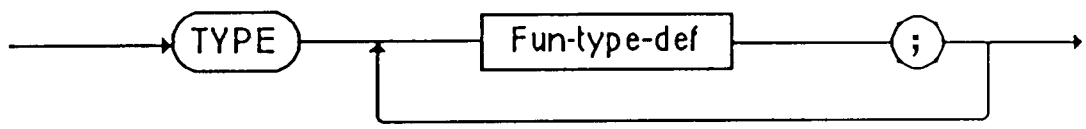
Dom-def



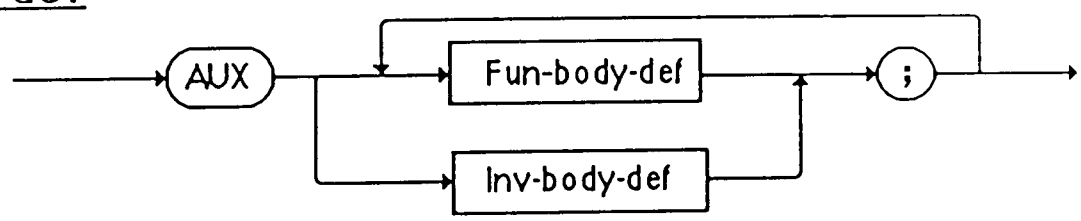
Adt-def



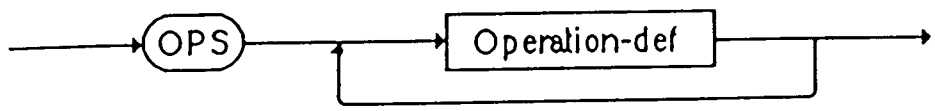
Type-def



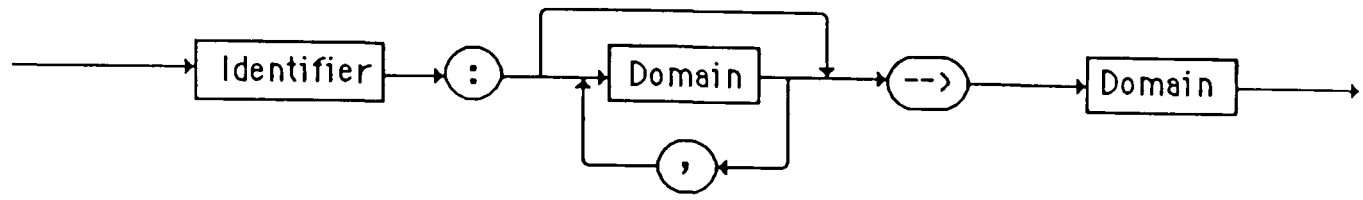
Aux-def



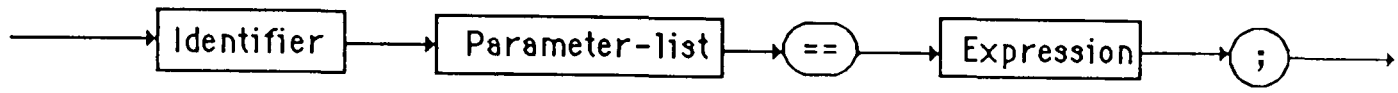
Ops-def



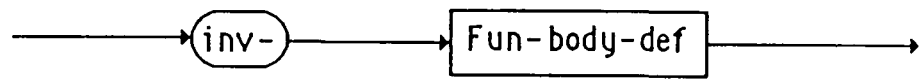
Fun-type-def



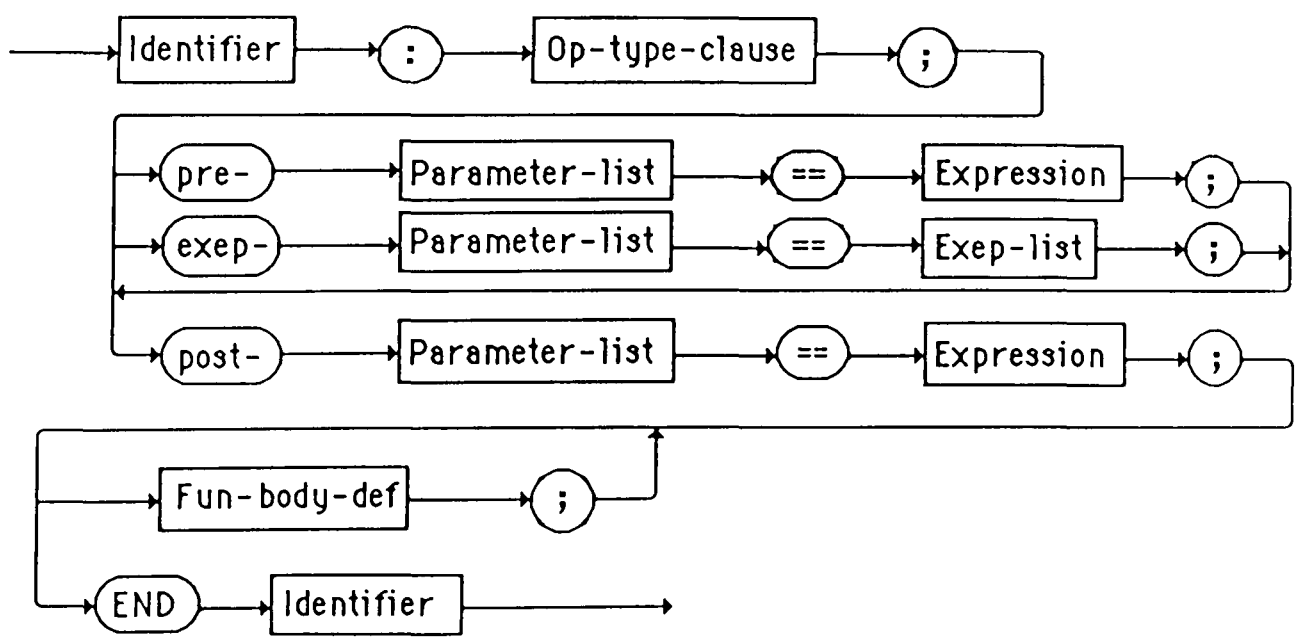
Fun-body-def



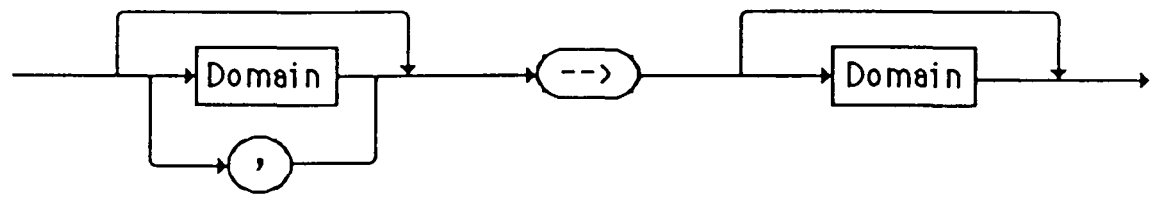
Inv-body-def



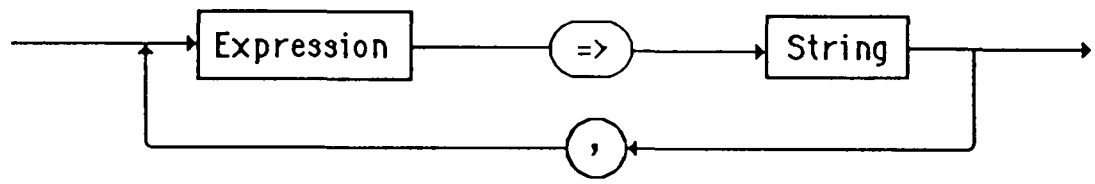
Operation-def



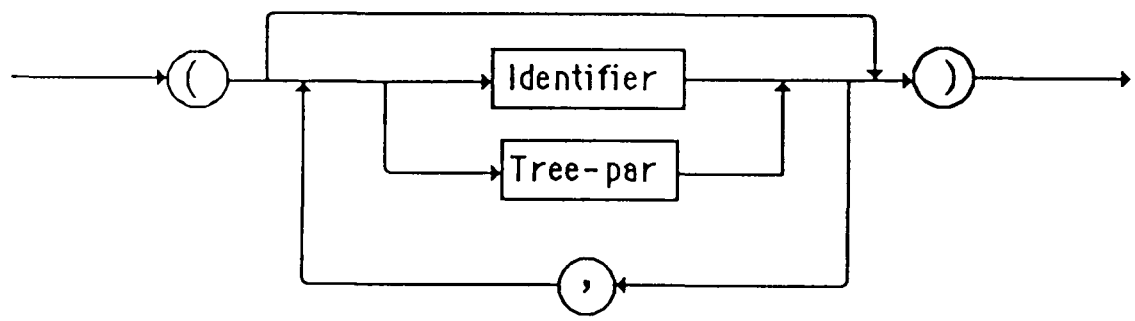
Op-type-clause



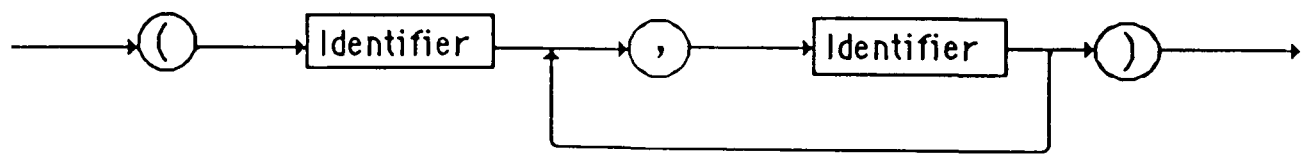
Exep-list



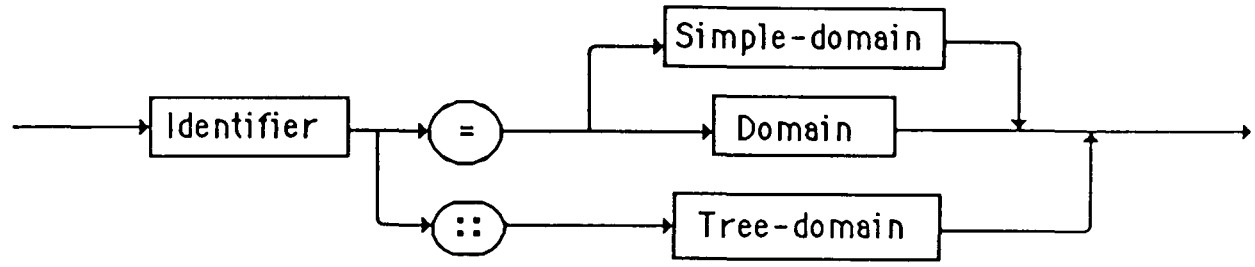
Parameter-list



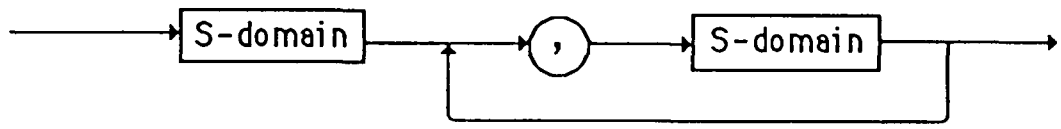
Tree-par



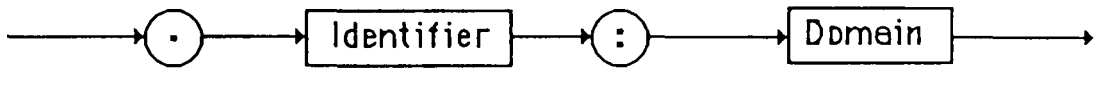
Abs-syn-rule



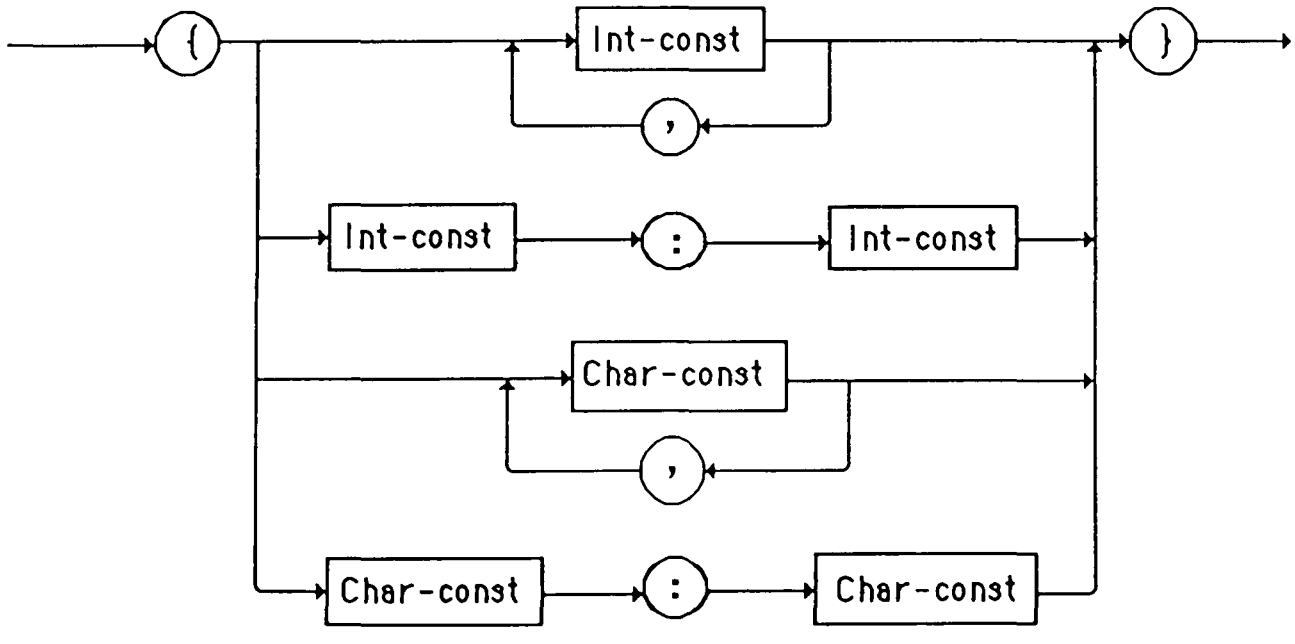
Tree-domain



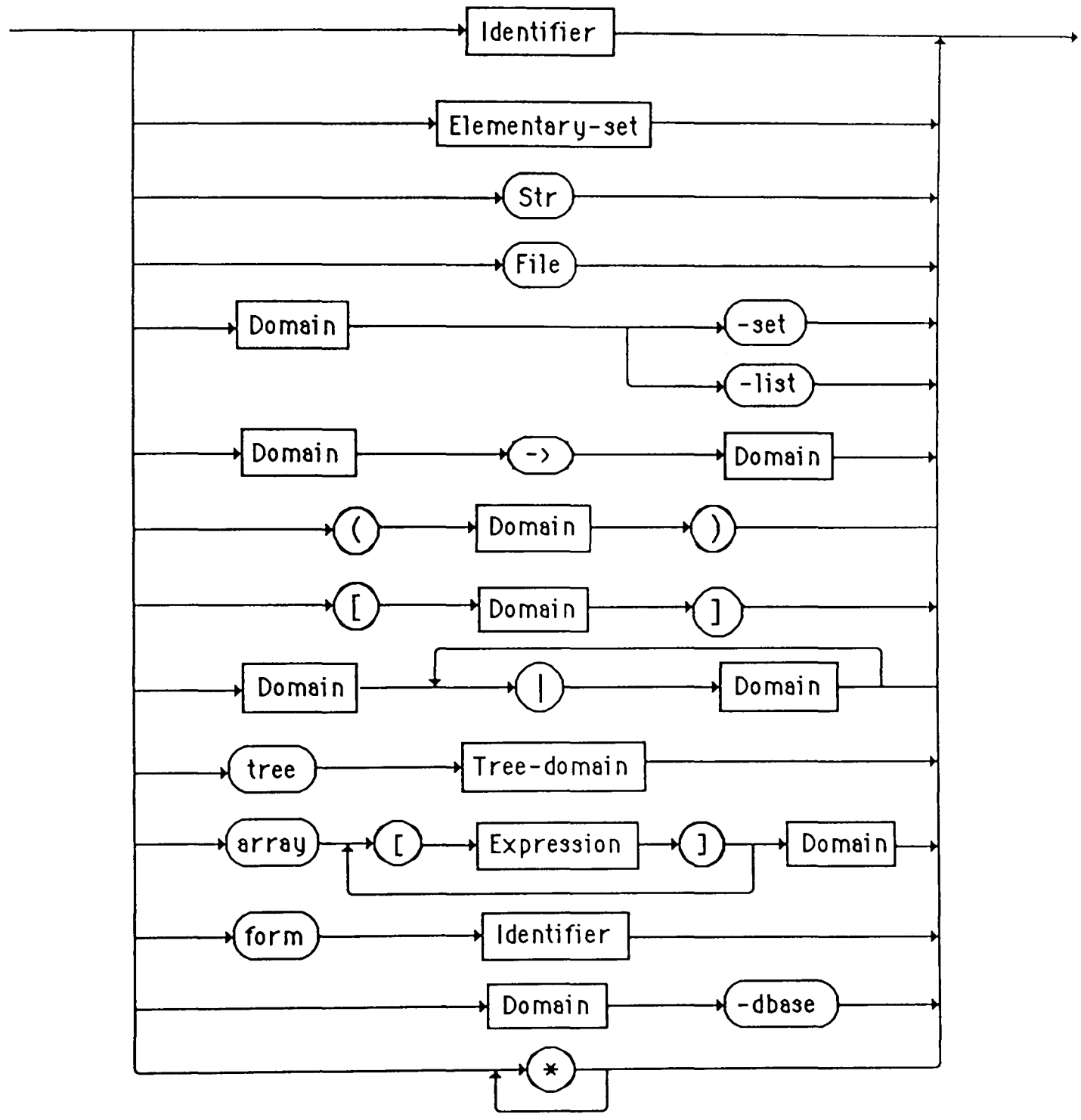
S-domain



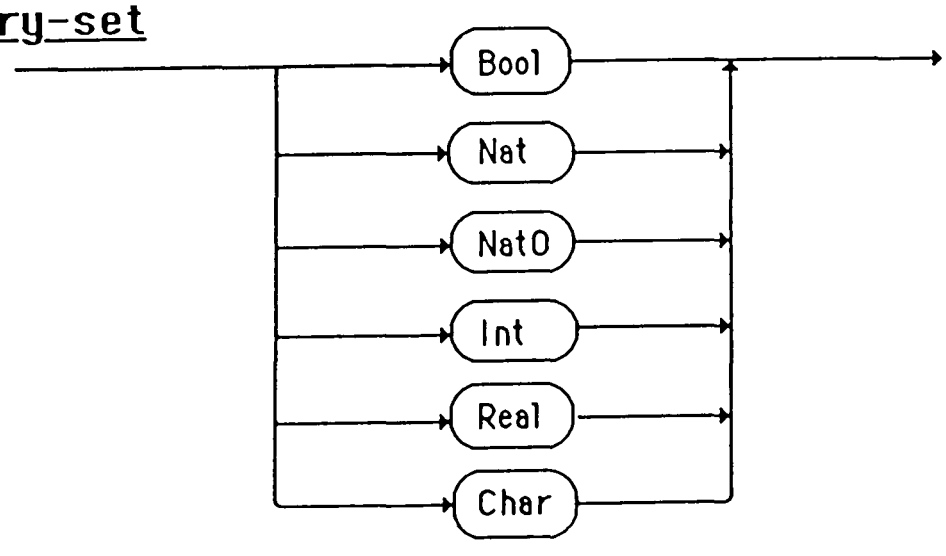
Simple-domain



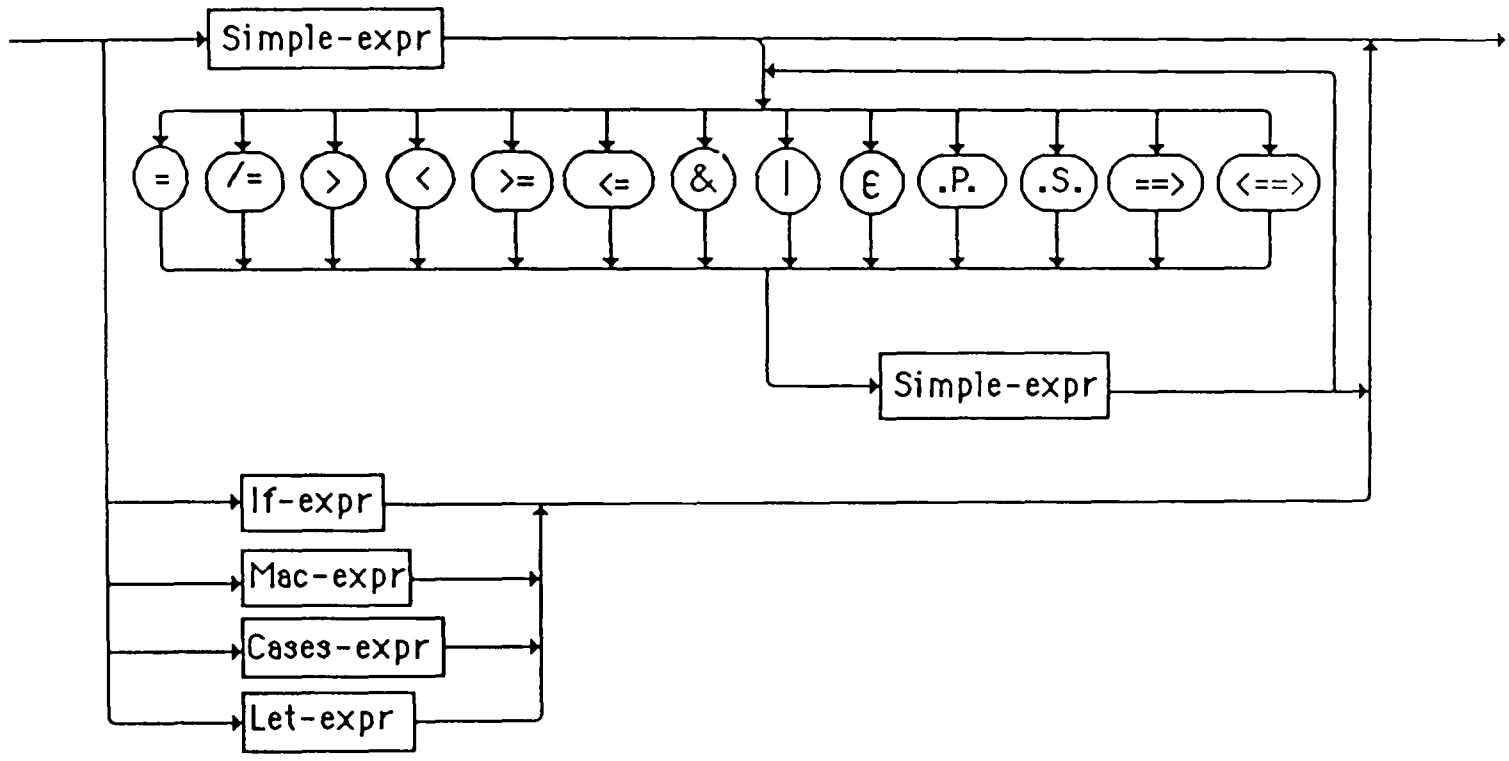
Domain



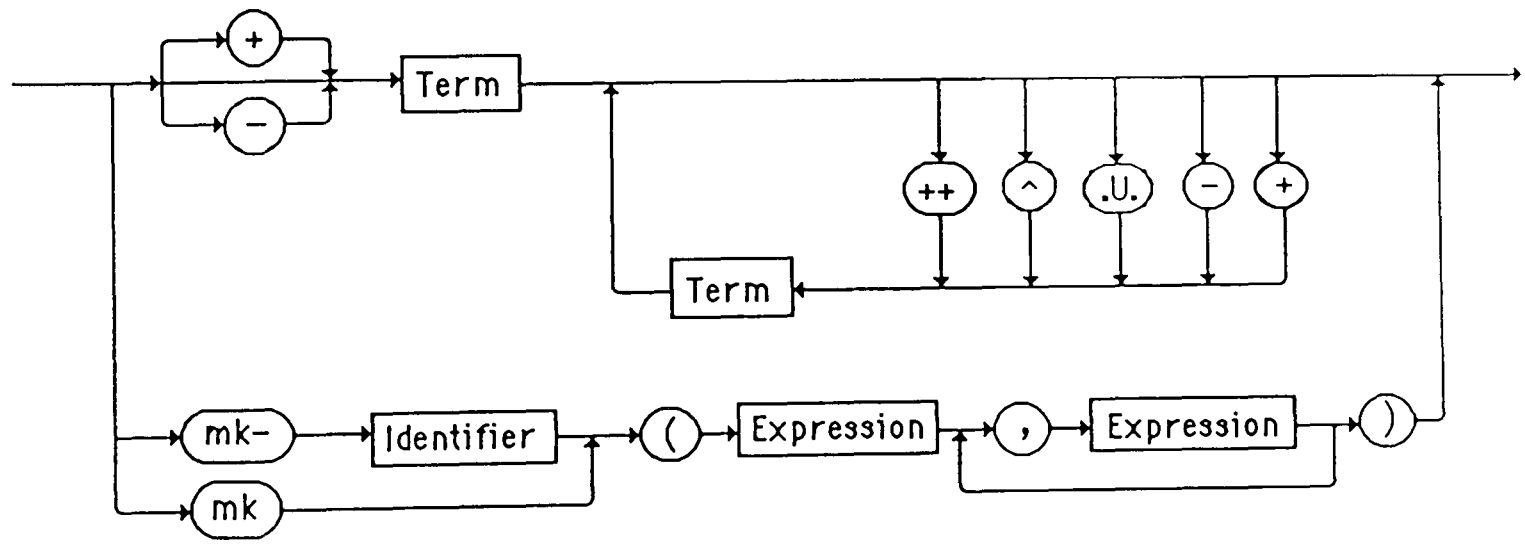
Elementary-set



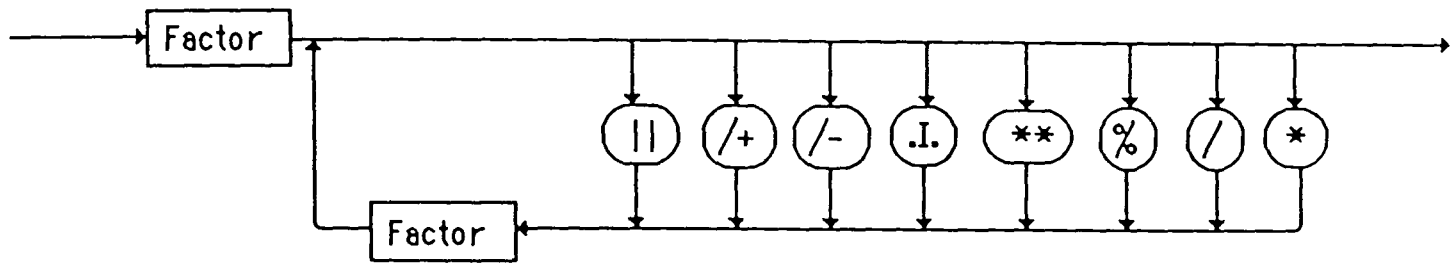
Expression



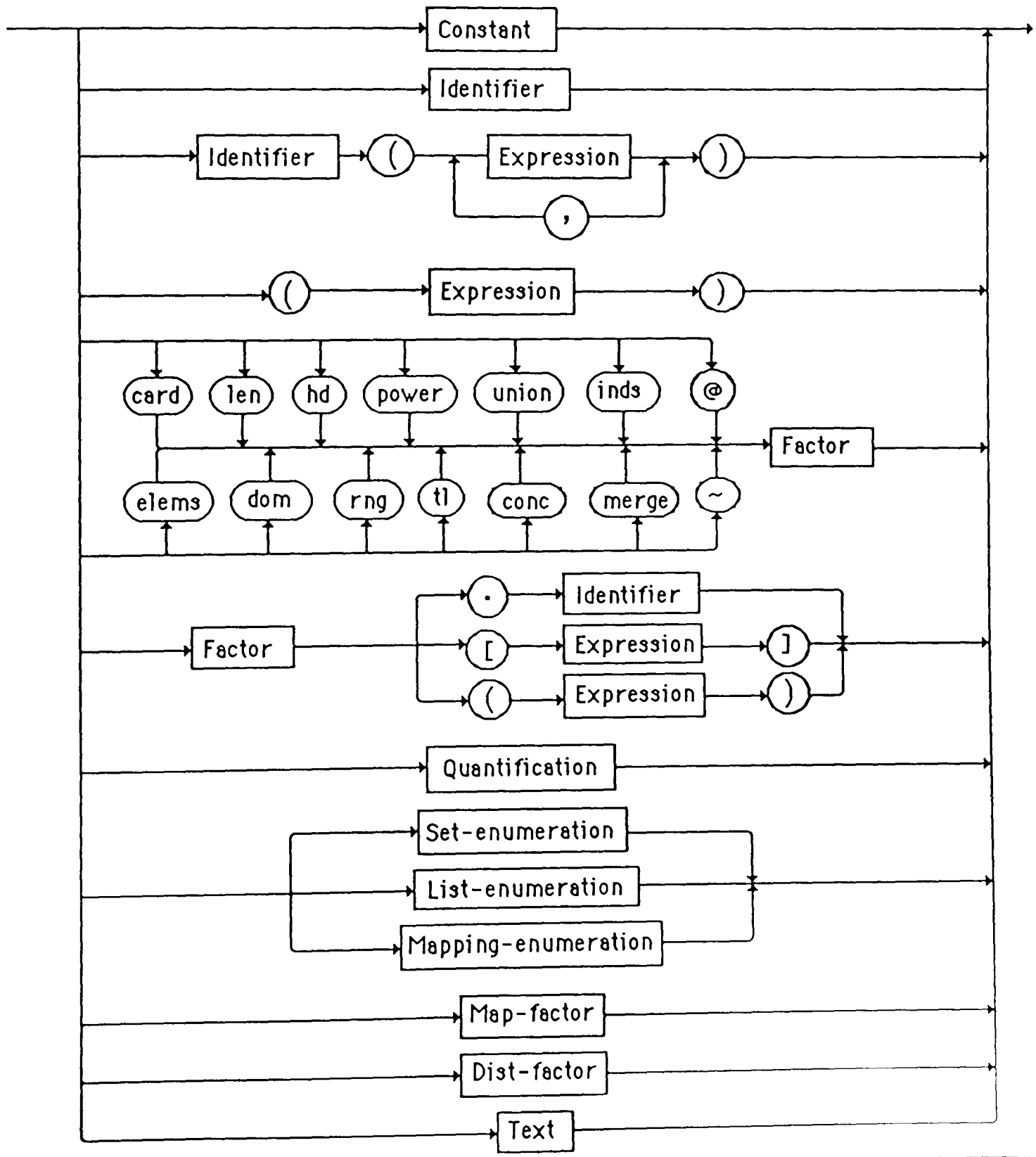
Simple-expr



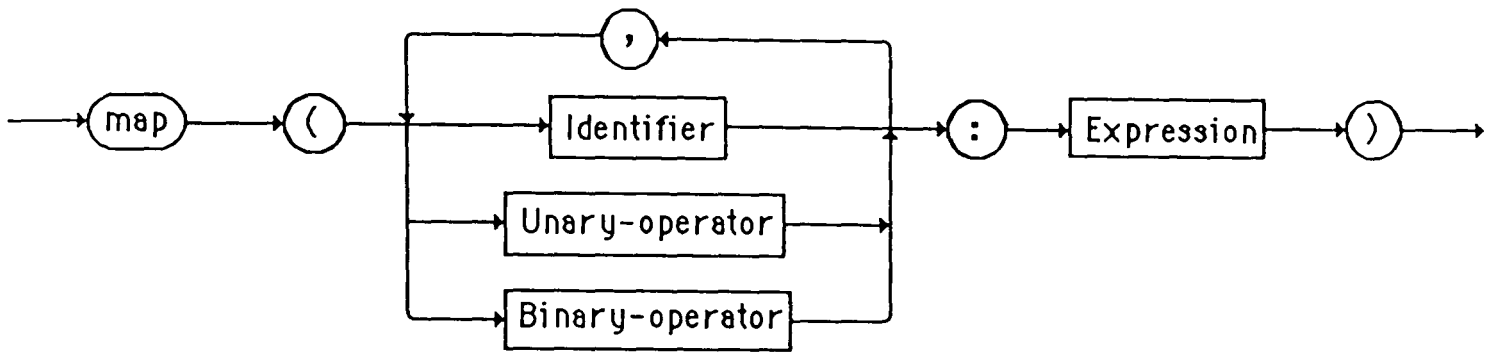
Term


Factor
||
/+
/-
.I.
**
%
/
*
Factor

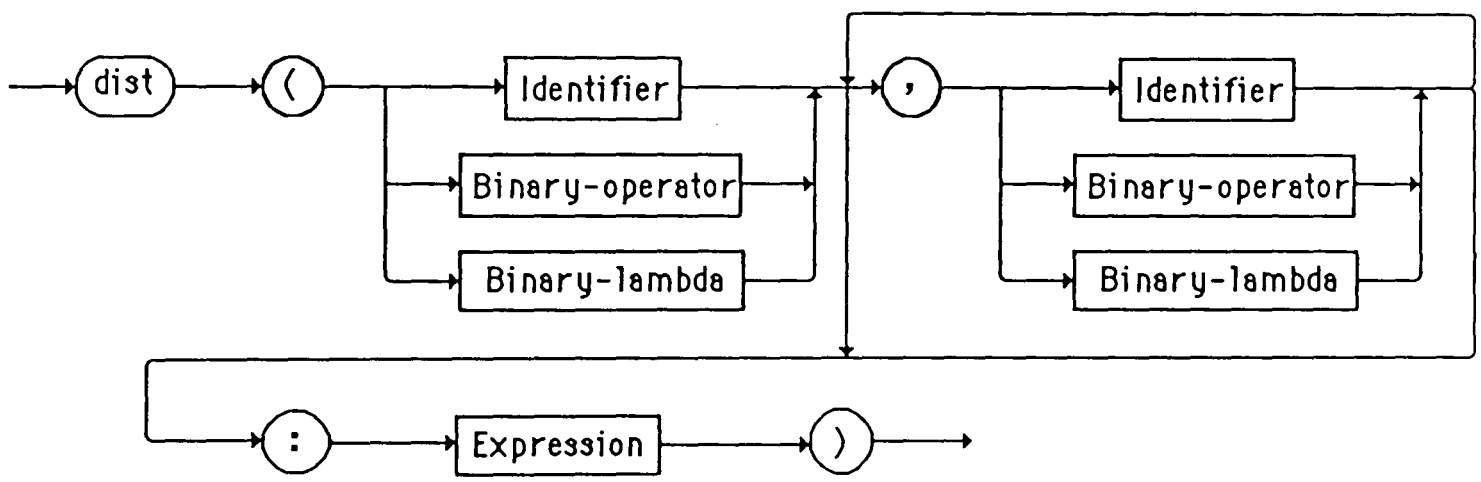
Factor


Constant
Identifier
Identifier
(
Expression
)
,
(
Expression
)
card
len
hd
power
union
inds
@
elems
dom
rng
tl
conc
merge
~
Factor
Factor
.
[
(
Identifier
Expression
Expression
]
)
Quantification
Set-enumeration
List-enumeration
Mapping-enumeration
Map-factor
Dist-factor
Text

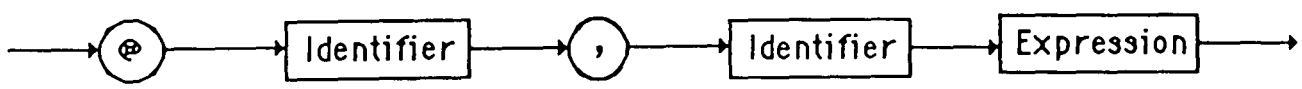
Map-factor



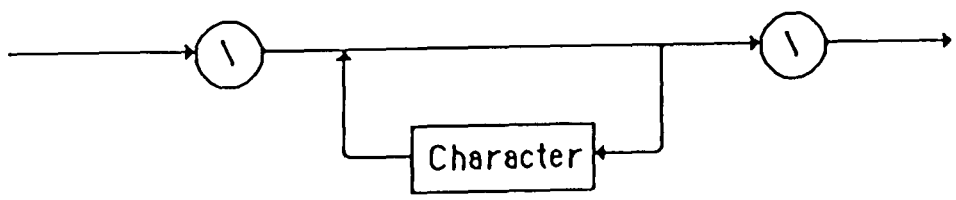
Dist-factor



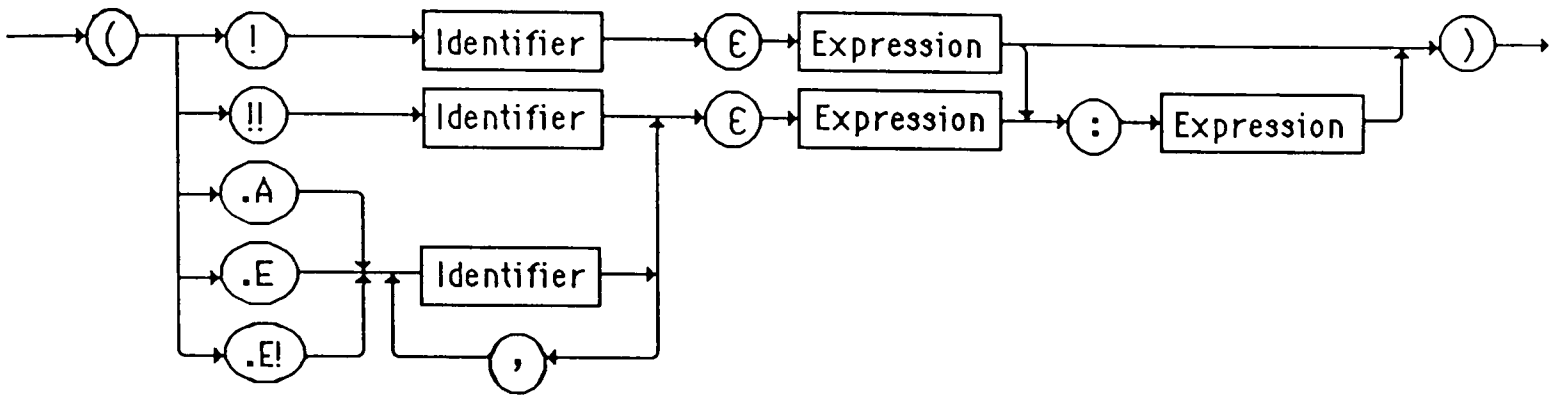
Binary-lambda



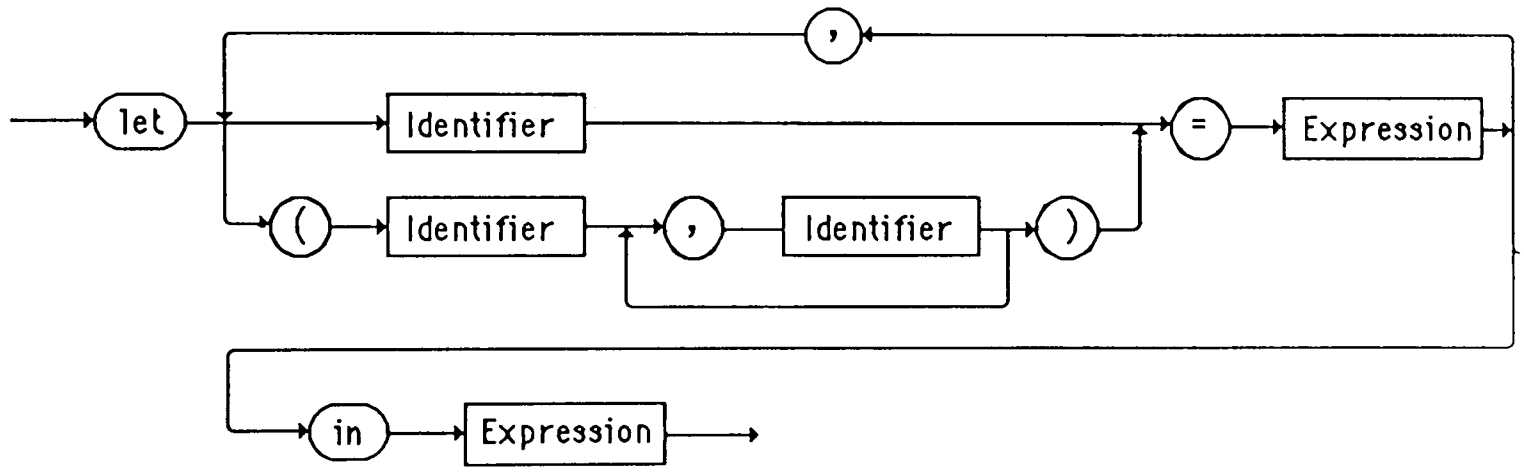
Text



Quantification

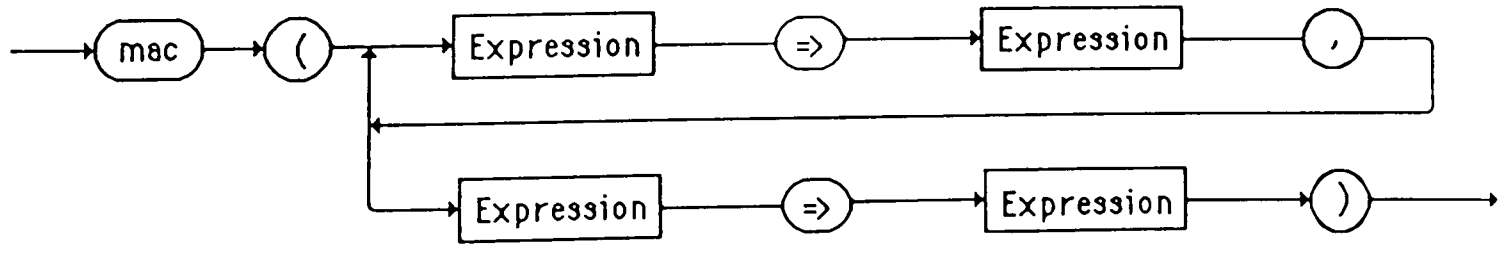


Let-expr

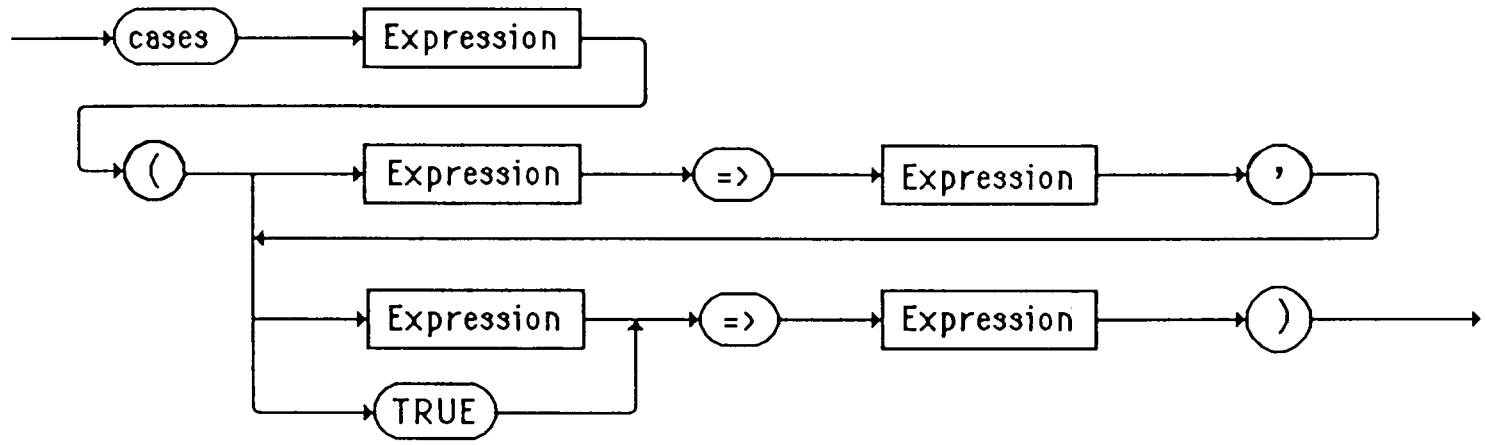


If-expr

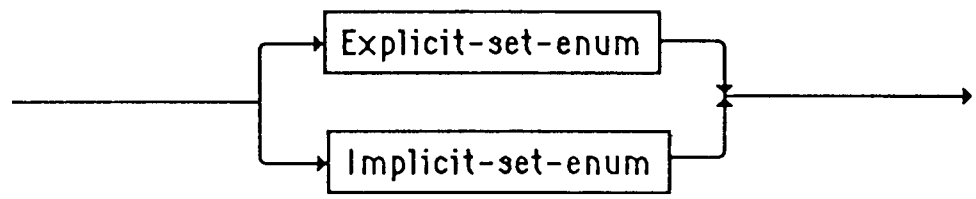
Mac-expr



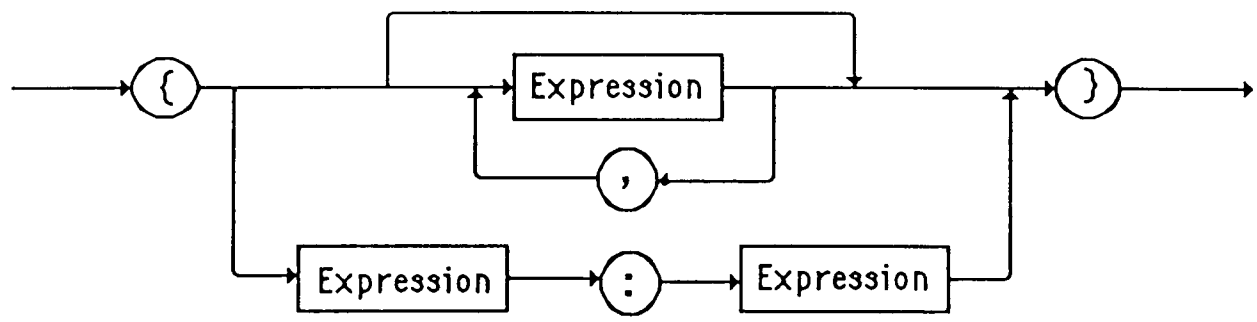
Cases-expr



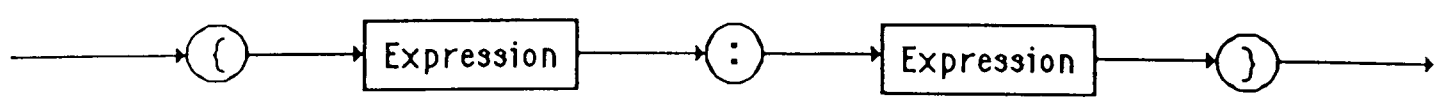
Set-enumeration



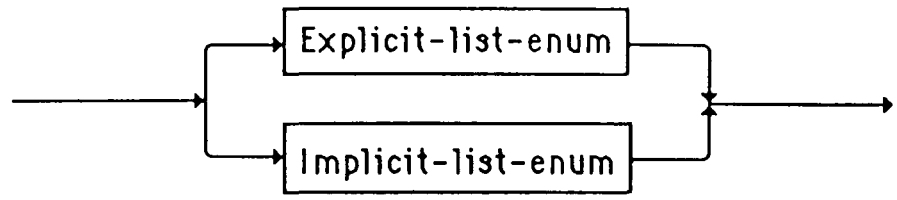
Explicit-set-enum



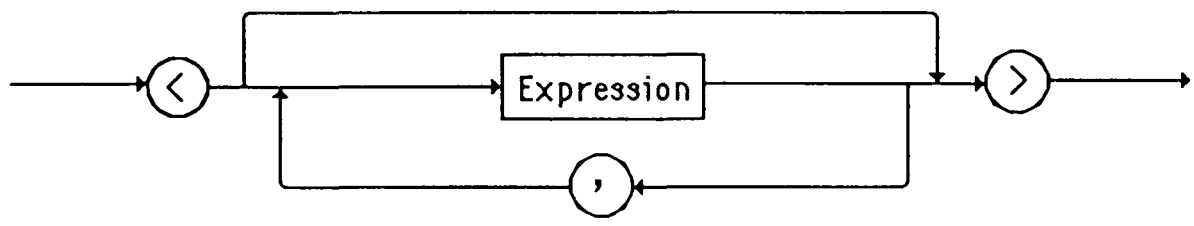
Implicit-set-enum



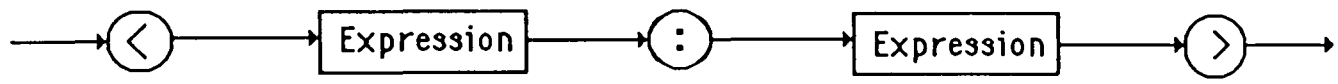
List-enum



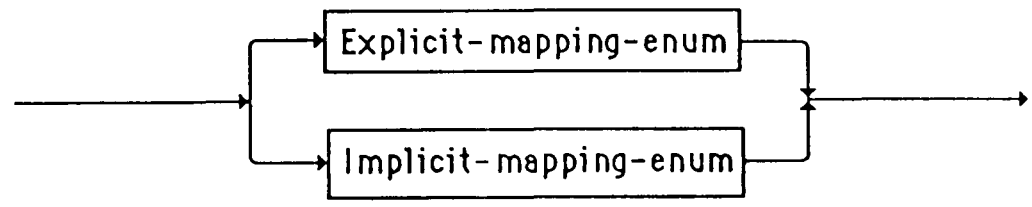
Explicit-list-enum



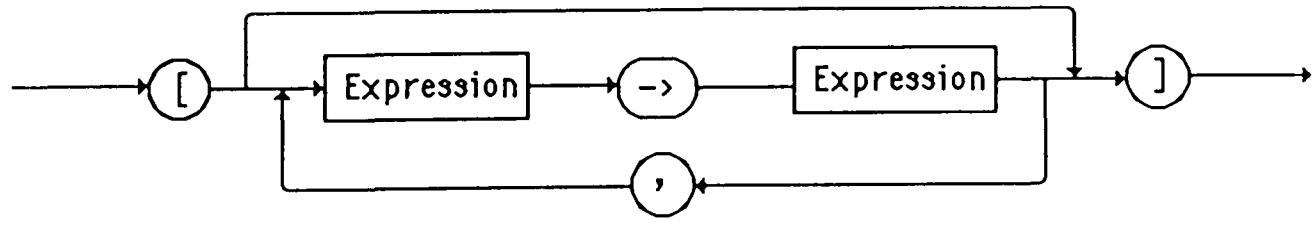
Implicit-list-enum



Mapping-eum

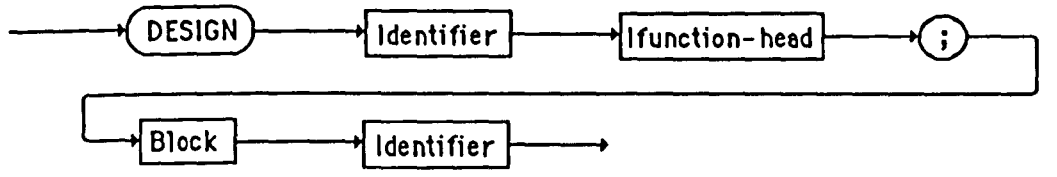


Explicit-mapping-enum

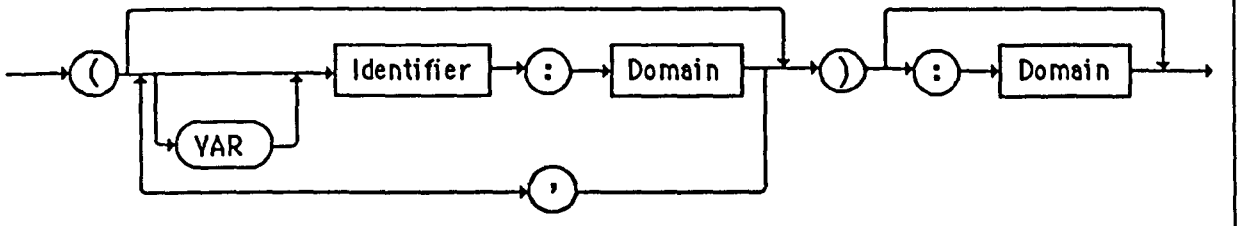


Implicit-mapping-enum

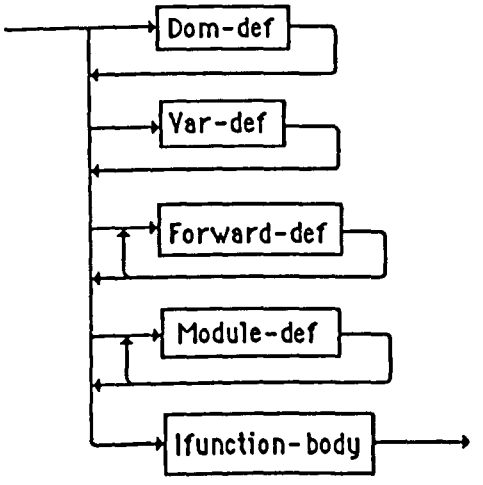
Design-def



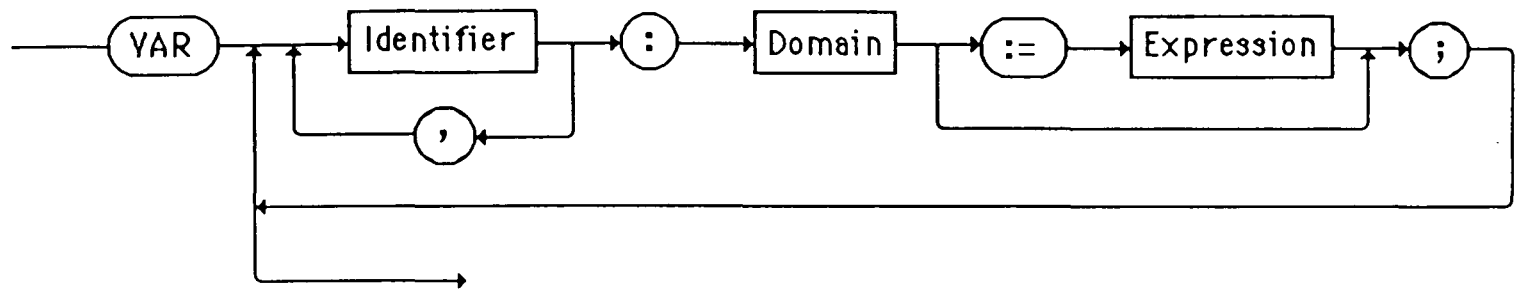
Ifunction-head



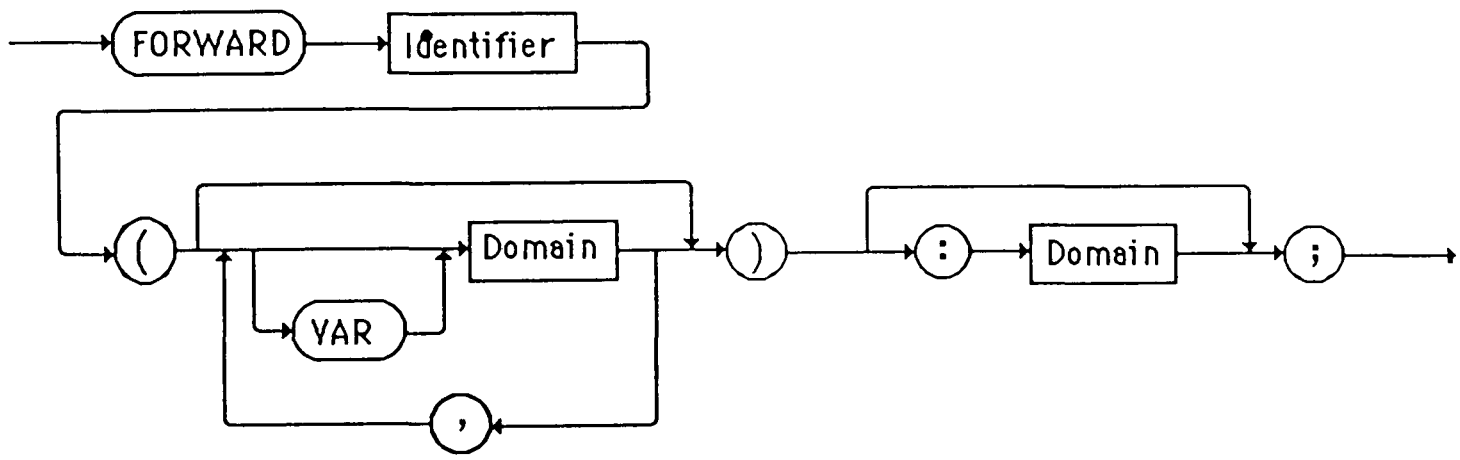
Block



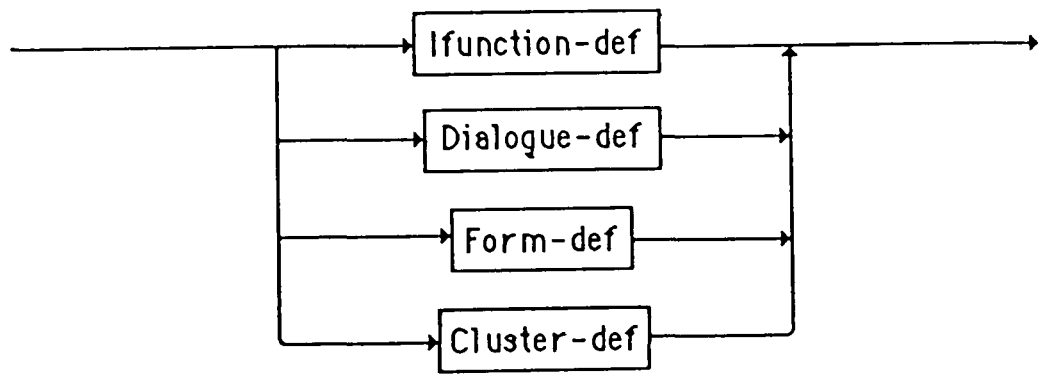
Var-def



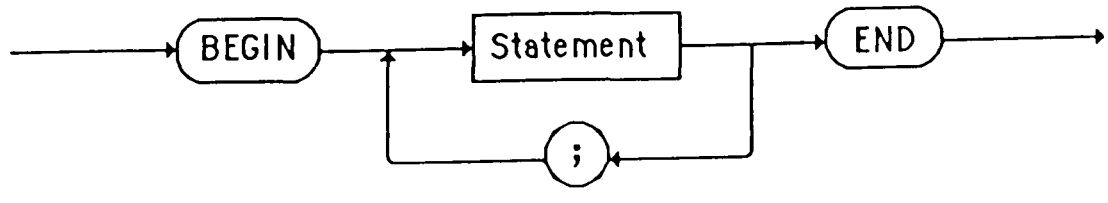
Forward-def



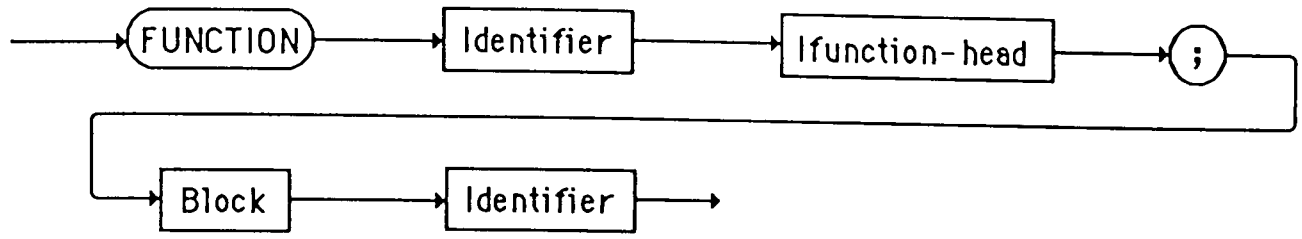
Module-def



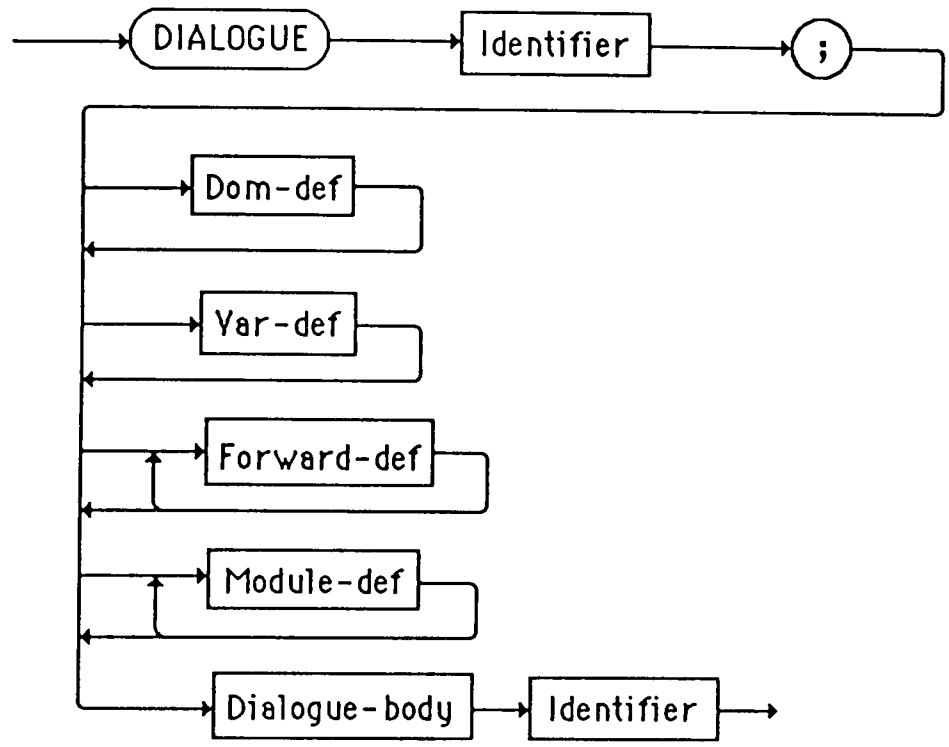
Ifunction-body



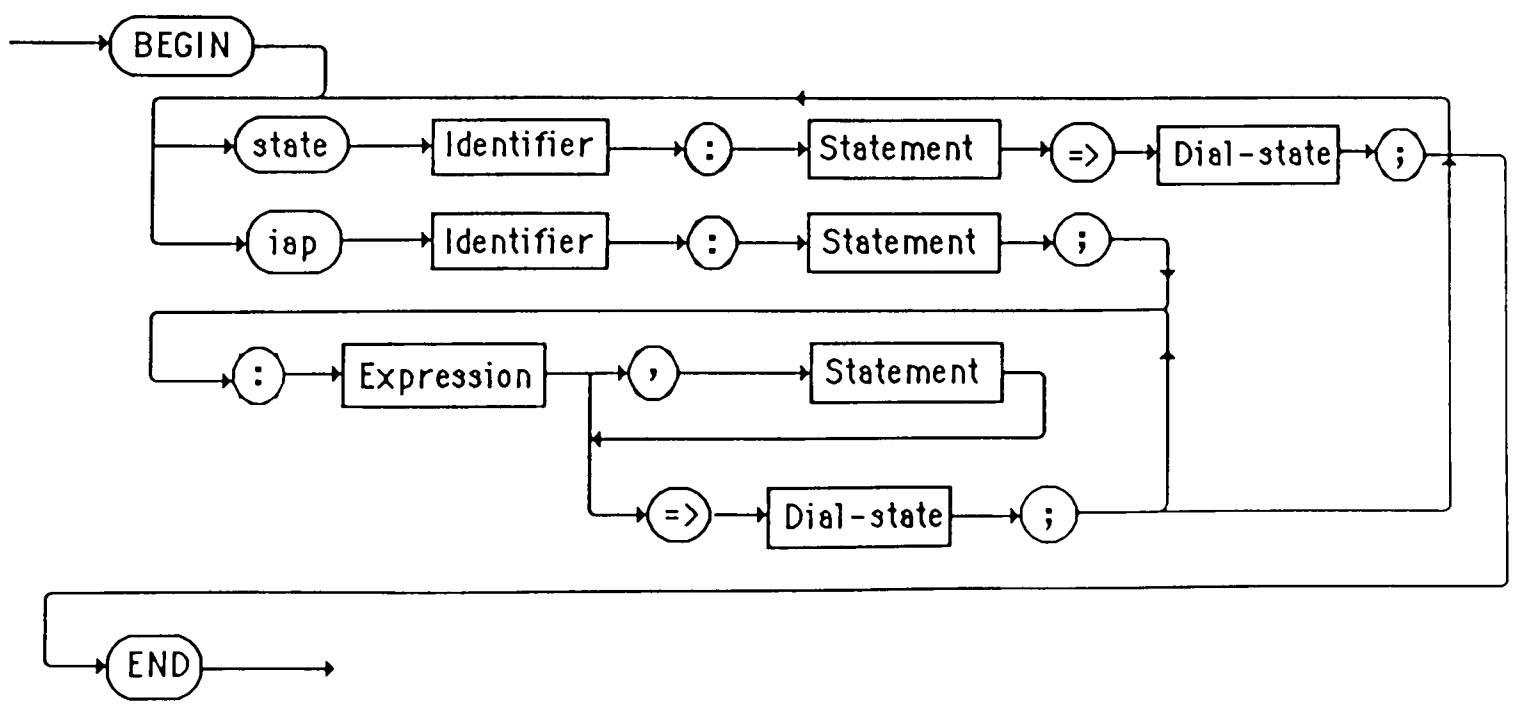
Ifunction-def



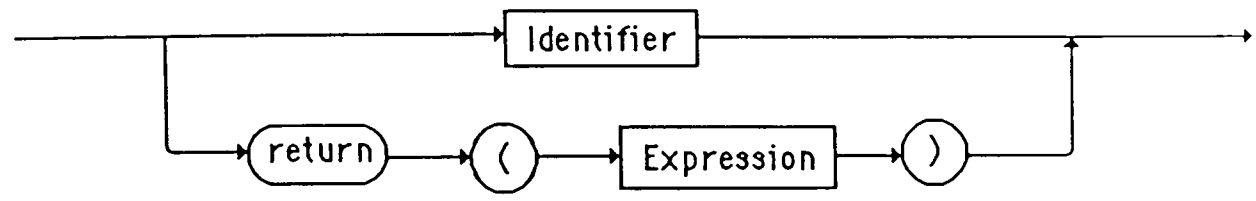
Dialogue-def



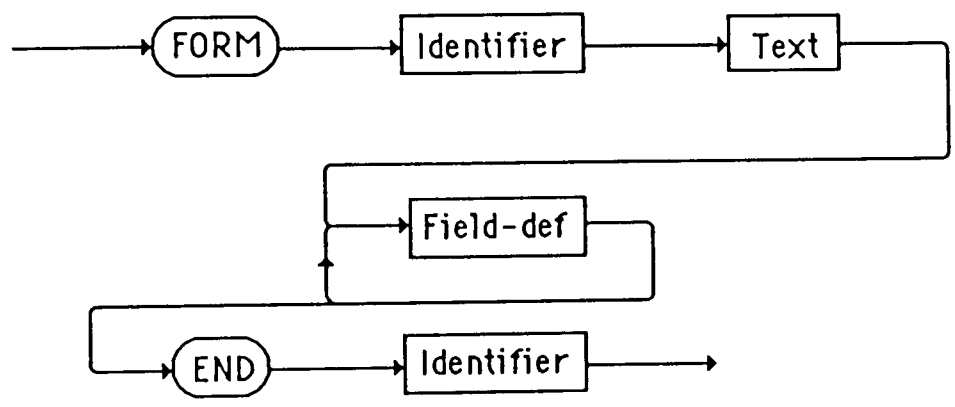
Dialogue-body



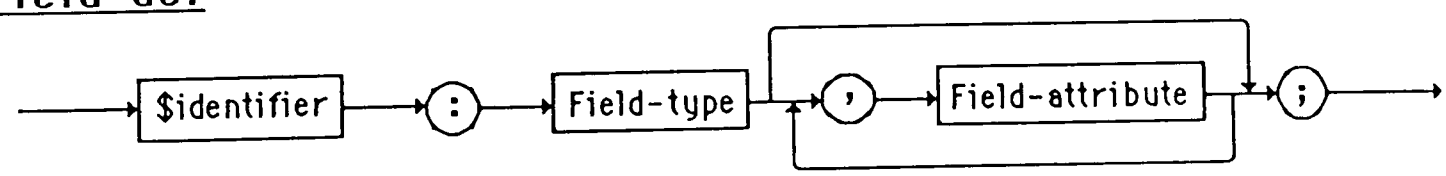
Dial-state



Form-def

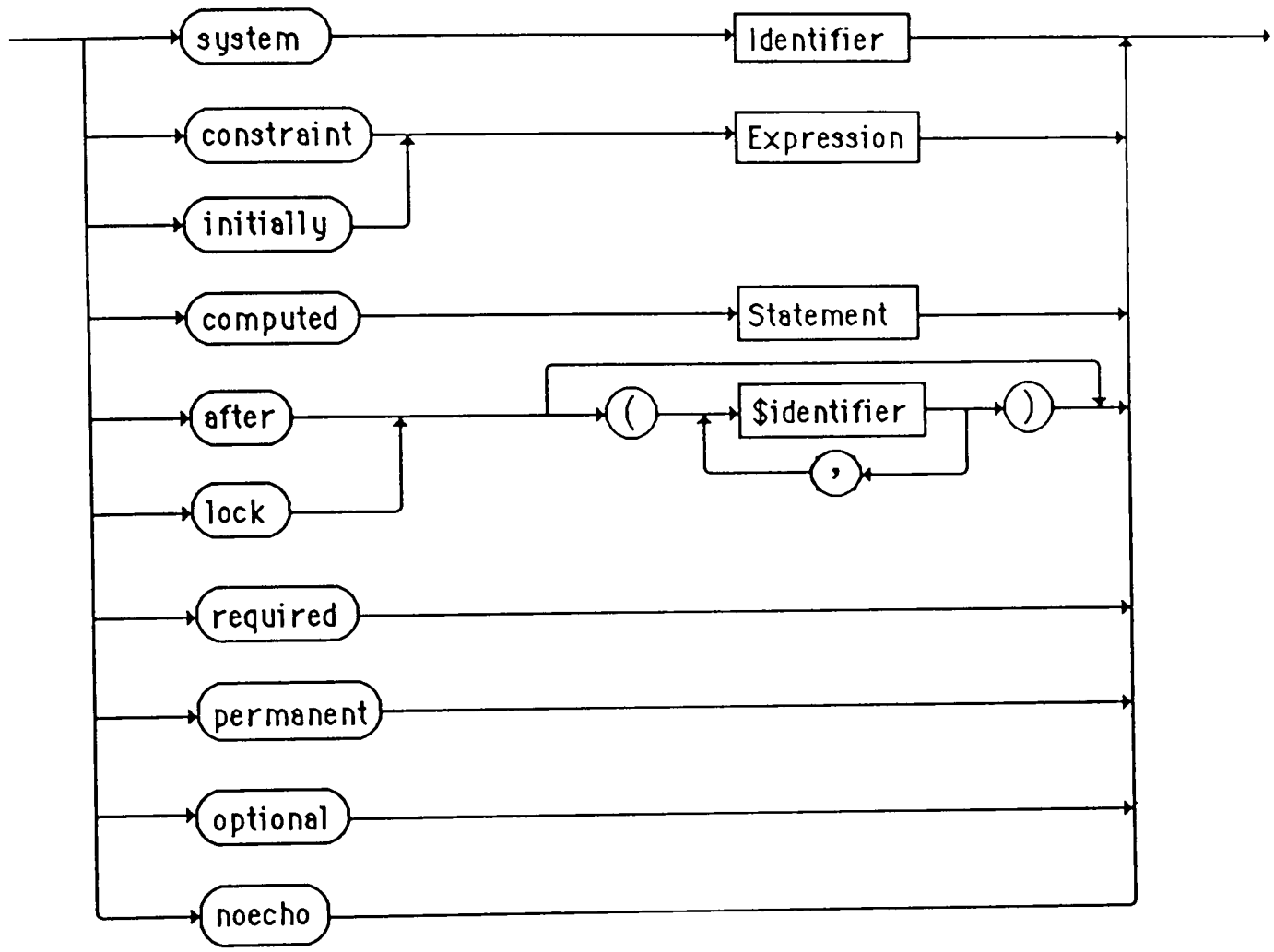


Field-def

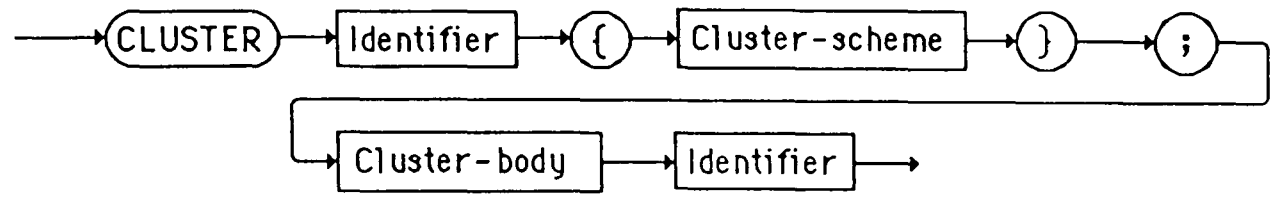


Field-type

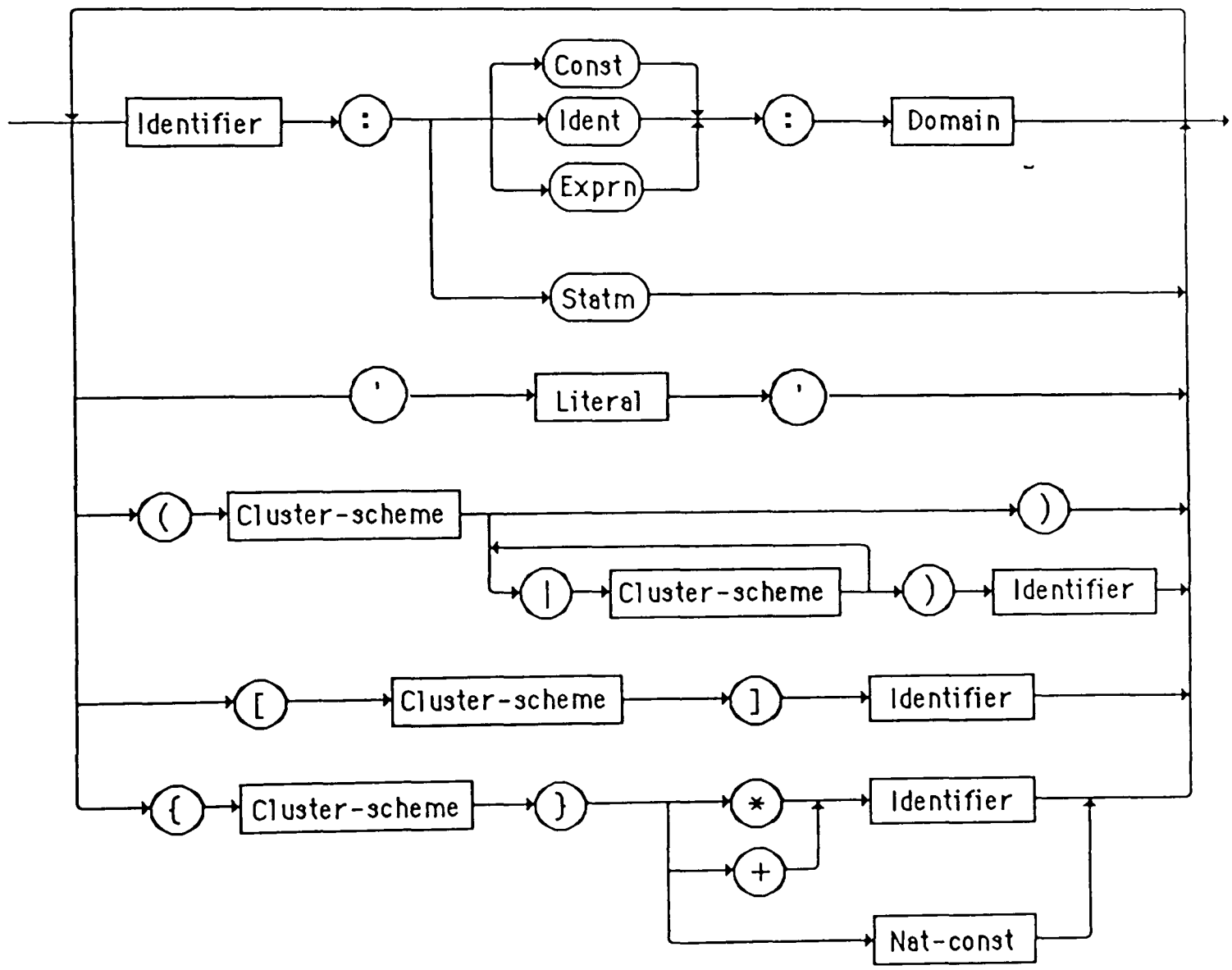
Field-attribute



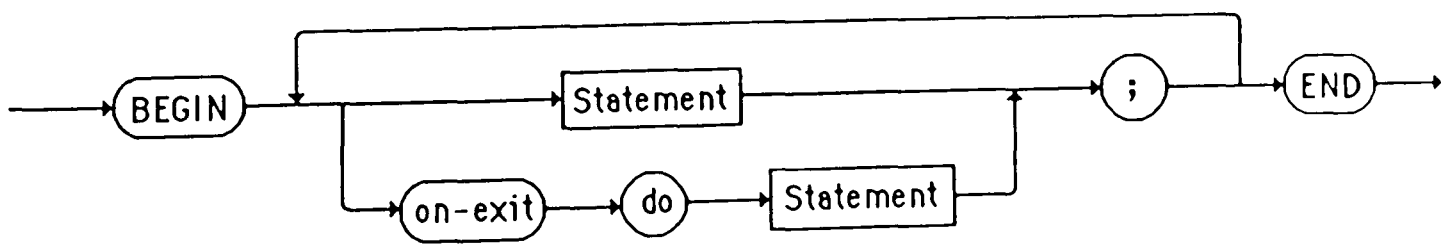
Cluster

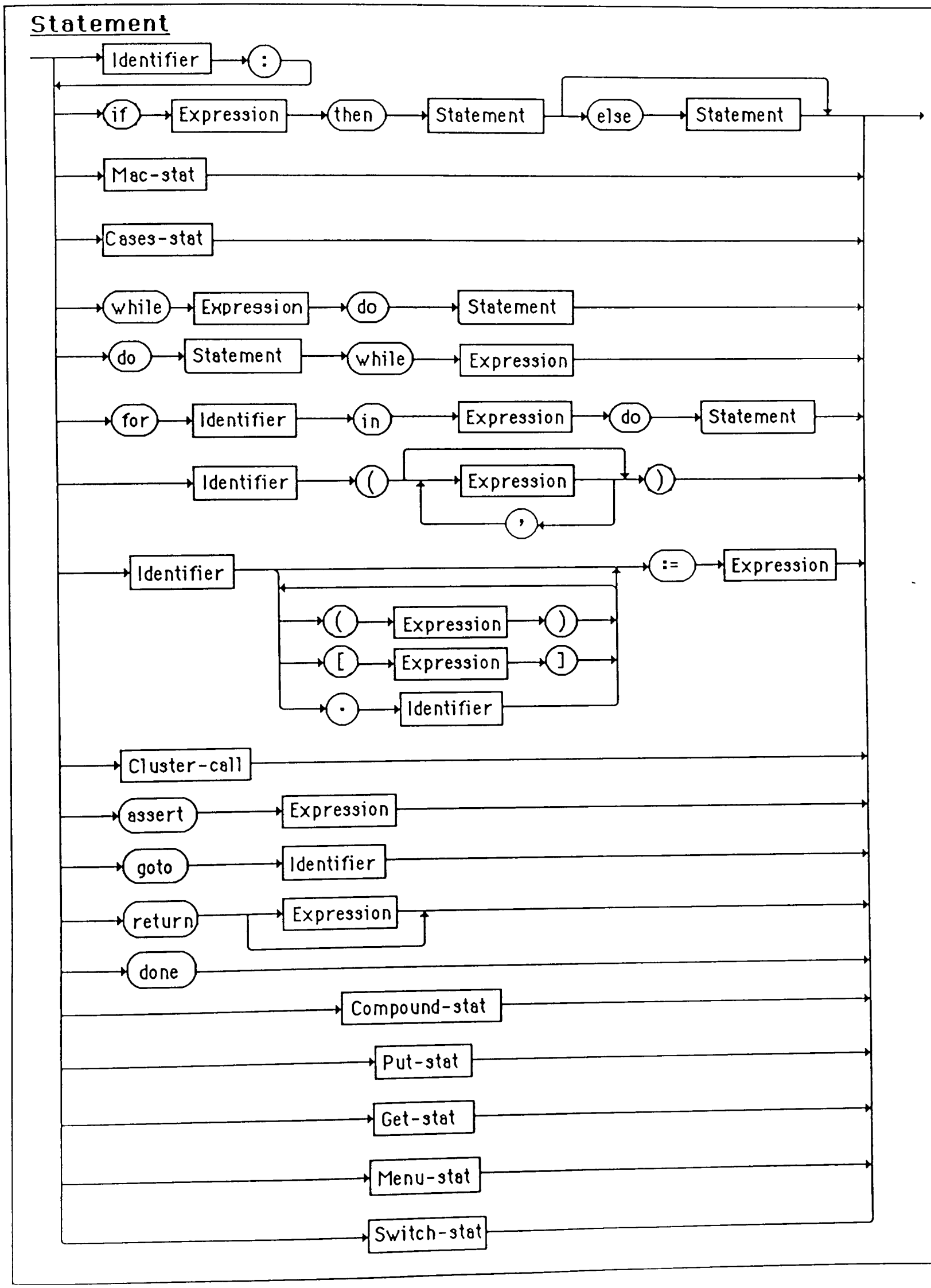


Cluster-scheme

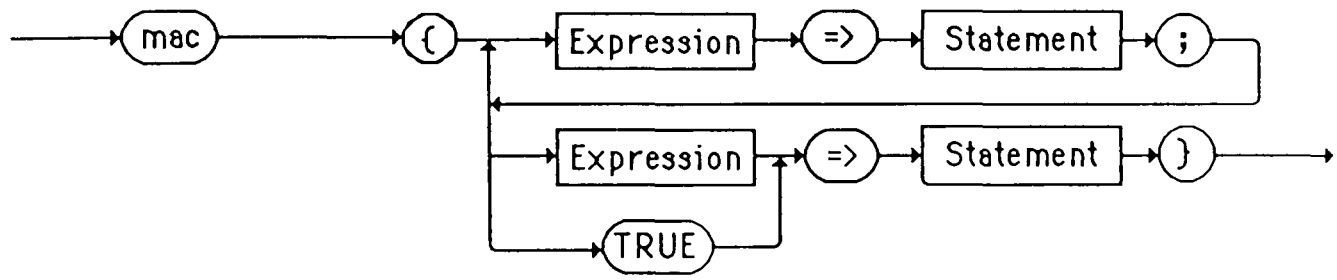


Cluster-body

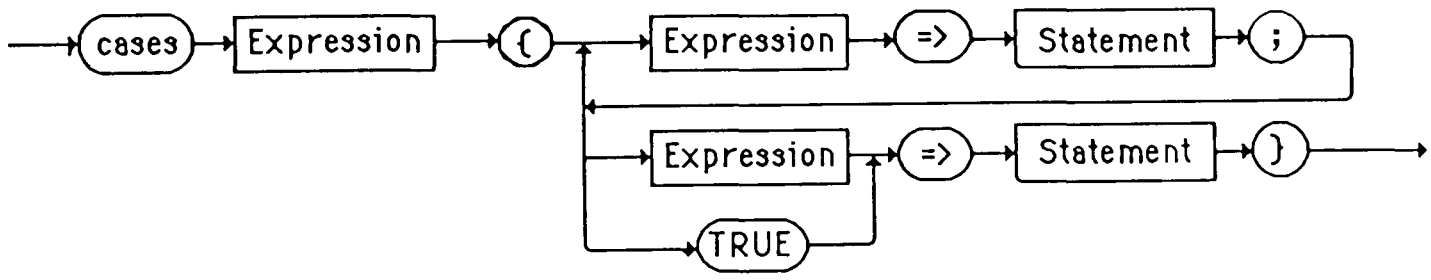



Statement
Identifier
:
if
Expression
then
Statement
else
Statement
Mac-stat
Cases-stat
while
Expression
do
Statement
do
Statement
while
Expression
for
Identifier
in
Expression
do
Statement
Identifier
(
Expression
,
)
Identifier
:=
Expression
(
Expression
)
[
Expression
]
.
Identifier
Cluster-call
assert
Expression
goto
Identifier
return
Expression
done
Compound-stat
Put-stat
Get-stat
Menu-stat
Switch-stat

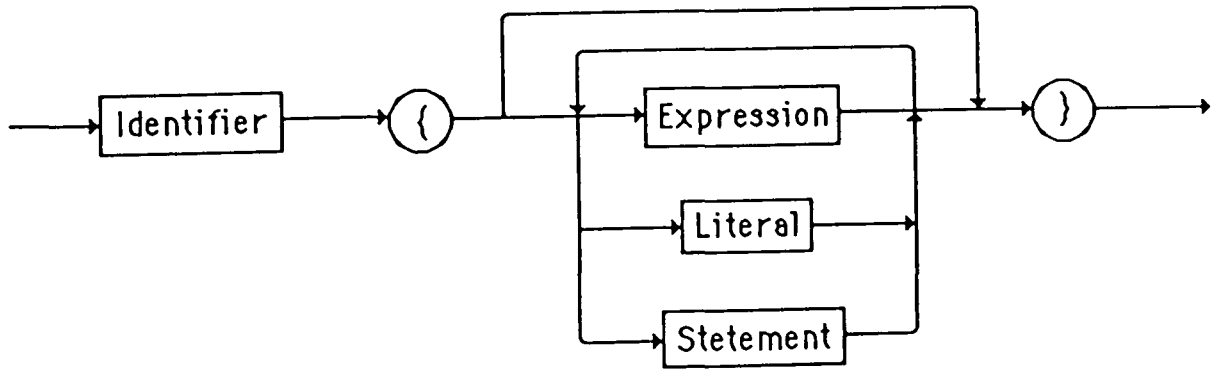
Mac-stat



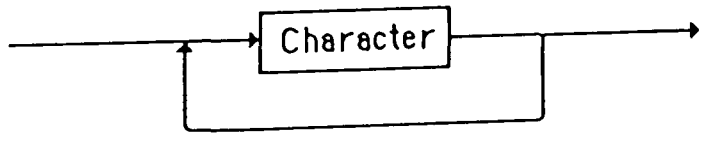
Cases-stat



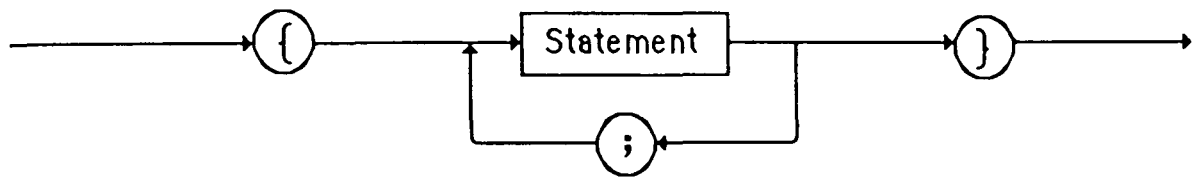
Cluster-call



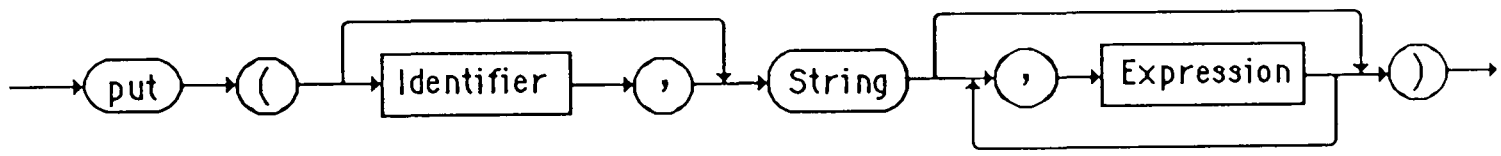
Literal



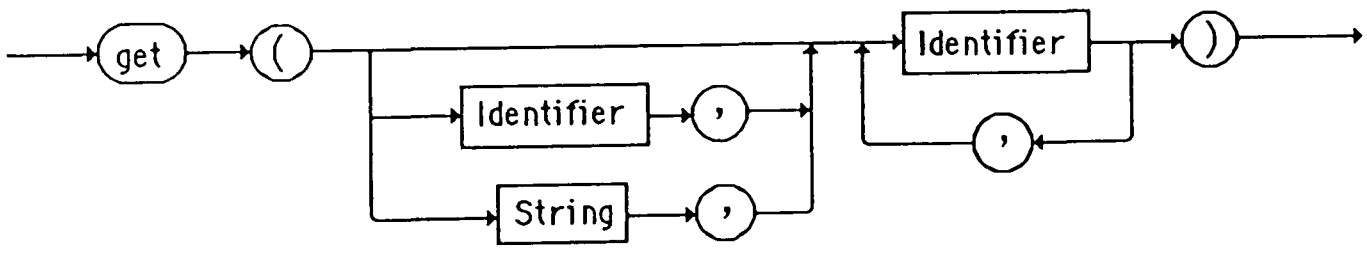
Compound-stat



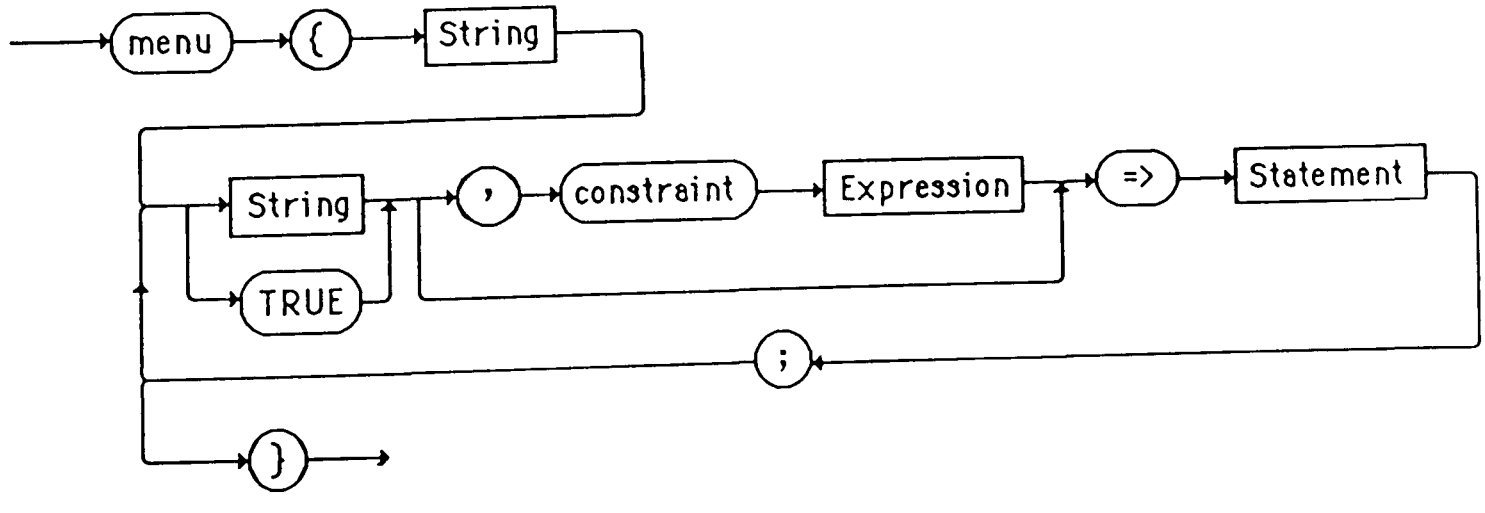
Put-stat



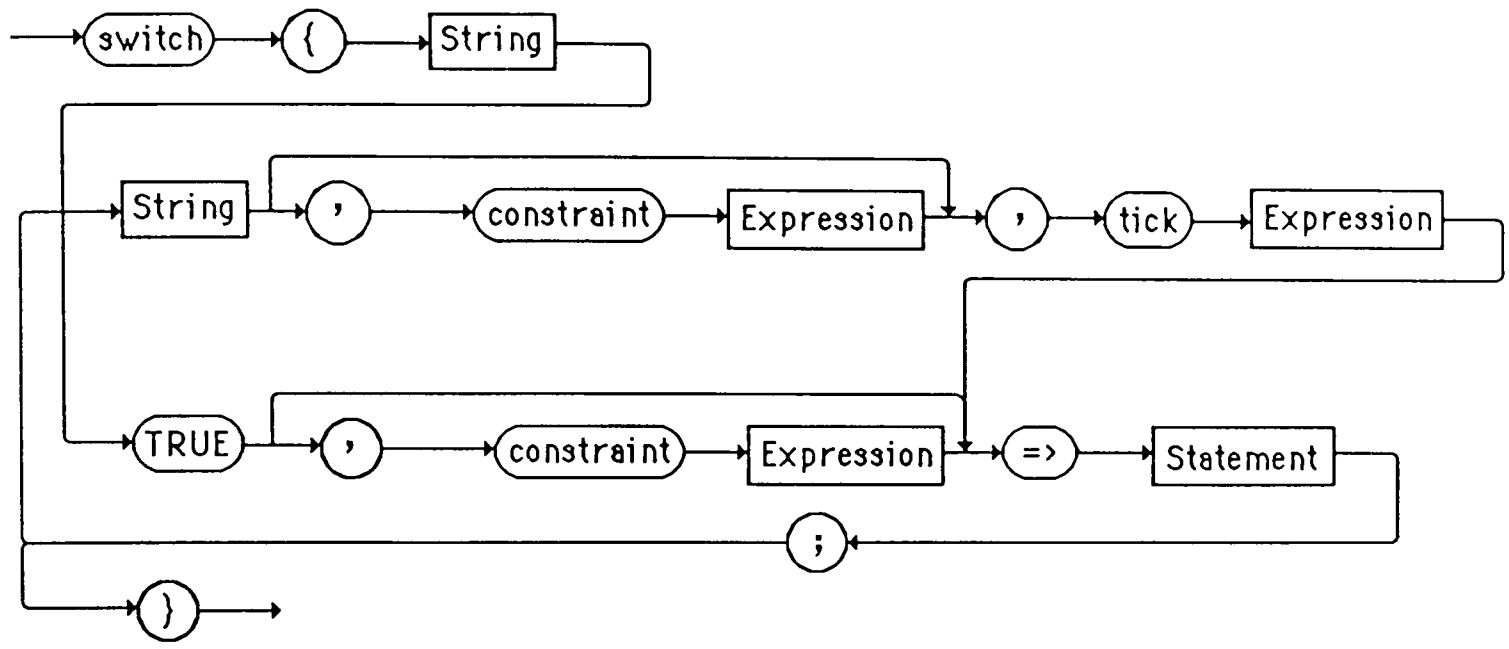
Get-stat



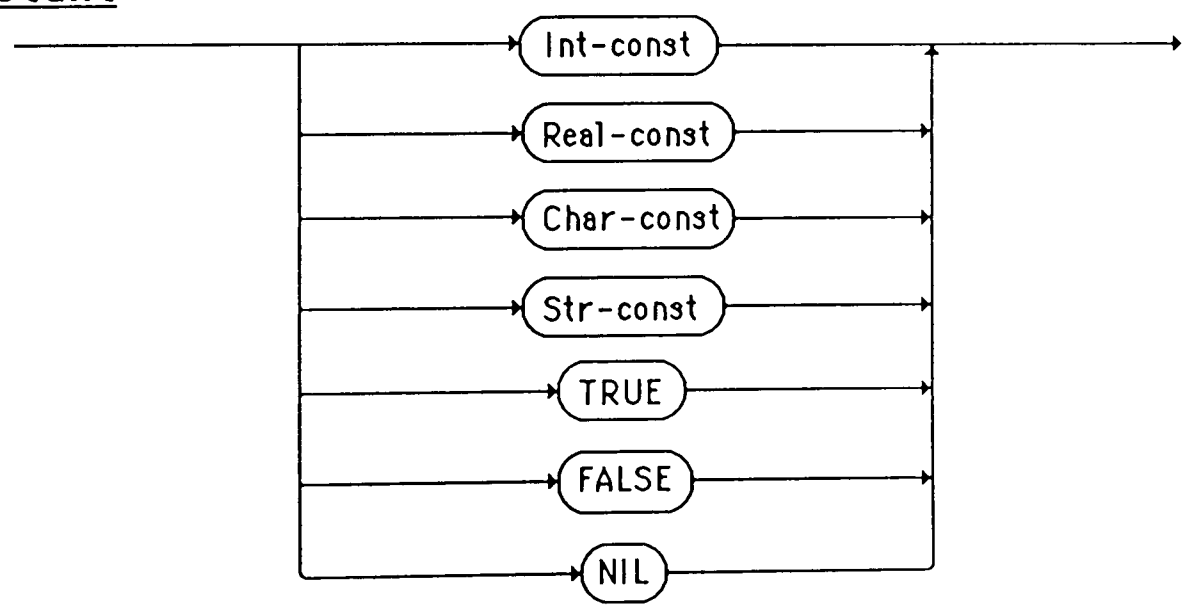
Menu-stat



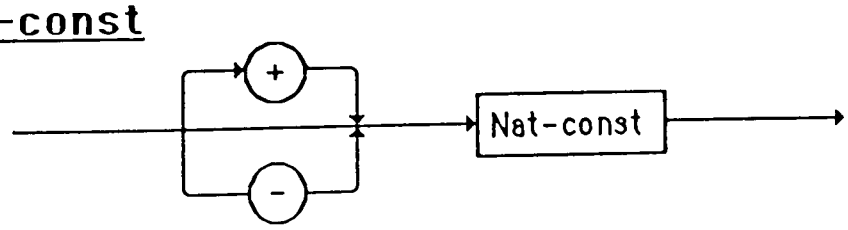
Switch-stat



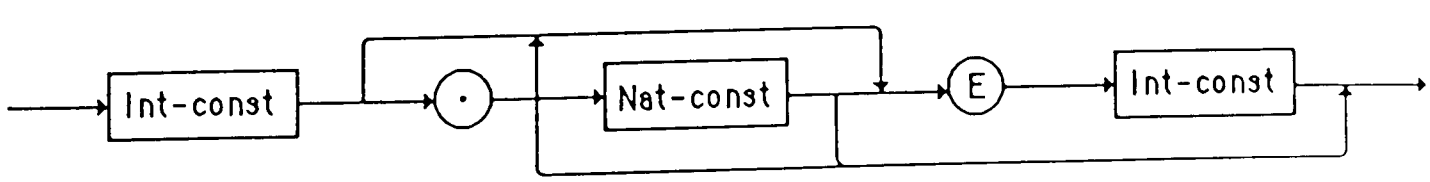
Constant



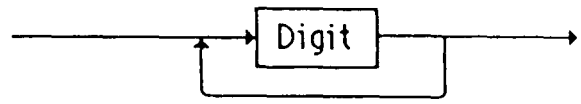
Int-const



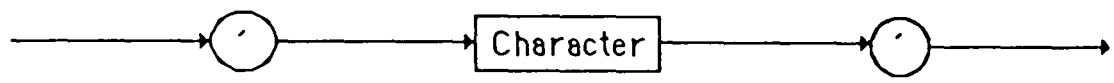
Real-const



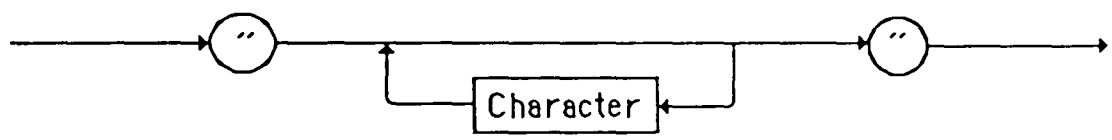
Nat-const



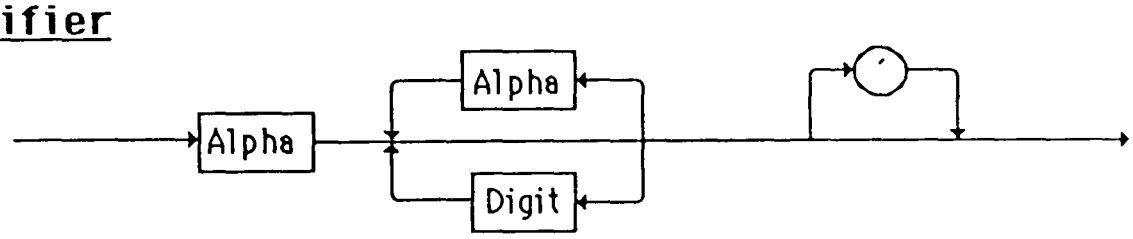
Char-const



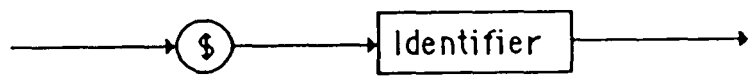
Str-const



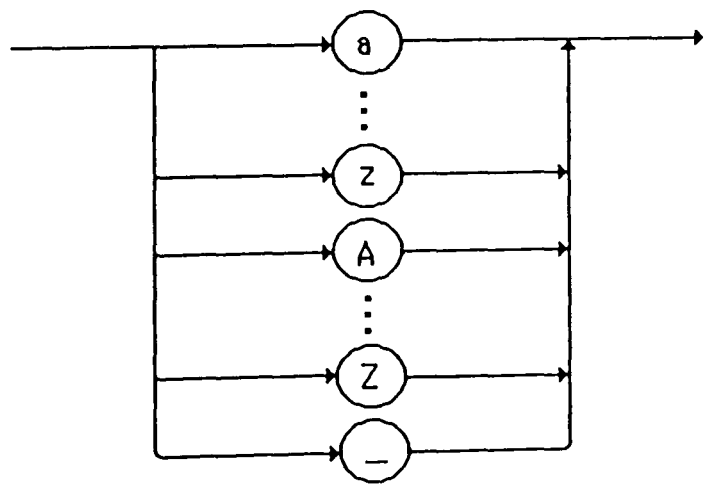
Identifier



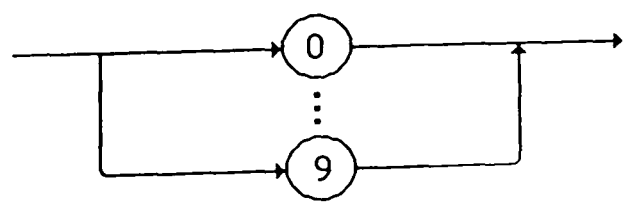
$identifier



Alpha



Digit



Character

Any visible character (i.e. Alpha, Digit, Space, Tab and special characters).

Appendix B COMPILATION EXAMPLE

This appendix illustrates the use of the EPROL compiler by listing the compilation of a simple stack specification which contains some deliberate errors. User input is printed in italics.

```
>> ec -t stack        /* compile file stack.e & produce compiler listing */
stack.e -ec-> stack.l, stack.t
     5         post(st,st') == st' := <>;
*EPROL_____________________________1
        1: ERROR 097, pre/post condition must be boolean.

     9         post(st,e,st') = st' = <i> || st;
*EPROL_________________________1________2
        1: ERROR 100, '==' expected.
        2: ERROR 115, identifier not bound.

-- 3  errors.
-- no warnings.

>> !cat stack.t       /* list the compiler listing file stack.t */
     0 ec -t stack

     1  ADT Stack
     2    DOM Stack = Int-list;
     3    OPS
     4      INIT: --> ;
     5        post(st,st') == st' := <>;
*EPROL_____________________________1
        1: ERROR 097, pre/post condition must be boolean.

     6      END INIT
     7
     8      PUSH: Int --> ;
     9        post(st,e,st') = st' = <i> || st;
*EPROL_________________________1________2
        1: ERROR 100, '==' expected.
        2: ERROR 115, identifier not bound.

    10      END PUSH
    11
    12      POP: --> ;
    13        pre(st) == st /= <>;
    14        post(st,st') == st' = tl st;
    15      END POP
    16 END Stack
-- 3  errors.
-- no warnings.
>>
```

Appendix C STANDARD LIBRARIES

Five standard libraries of EPROL are described. The use of each library must be explicitly stated using a library directive (e.g. `%library "scr"`.)

C.1 math

```
exp: Real --> Real;
exp(x)
```
Returns the number e raised to the power of x.

```
log: Real --> Real;
log(x)
```
Returns the (base e) logarithm of x.

```
fix: Real --> Int;
fix(x)
```
Returns the integral part of x.

```
float: Int --> Real;
float(i)
```
Converts i to a real number.

```
abs: Int | Real --> Int | Real;
abs(n)
```
Returns the absolute value of n.

```
sqrt: Int | Real --> Real;
sqrt(n)
```
Returns the square root of n.

```
sin: Real --> Real;
sin(x)
```
Returns the sine of angle x.

```
cos: Real --> Real;
cos(x)
```
Returns the cosine of angle x.

```
evenp: Int --> Bool;
evenp(i)
```
Returns TRUE if i is an even number and FALSE otherwise.

```
oddp: Int --> Bool;
oddp(i)
```
Returns TRUE if i is an odd number and FALSE otherwise.

C.2 str

```
st_new: Nat0 --> Str;
st_new(i)
```
Returns a new string which initially contains i blanks.

```
st_len: Str --> Nat0;
st_len(s)
```

Returns the length (i.e. the number of characters) of `s`.

`st_app: Str, Str --> Str;`
`st_app(s1,s2)`
Returns a new string which is the result of appending `s2` to `s1`.

`st_left: Nat0, Str --> Str;`
`st_left(i,s)`
Returns a new string which consists of the `i` leftmost characters of `s`.

`st_right: Nat0, Str --> Str;`
`st_right(i,s)`
Returns a new string which consists of the `i` rightmost characters of `s`.

`st_mid: Nat0, Nat0, Str --> Str;`
`st_mid(i,j,s)`
Returns a new string which consists of the `i`-th through to the `j`-th character of `s`.

`st_mk: Char-list --> Str;`
`st_mk(cl)`
Returns a new string which consists of the characters in list `cl`.

`st_unmk: Str --> Char-list;`
`st_unmk(s)`
Returns a list of all characters in `s`.

C.3 io

`f_open (f: Str, m: Str): File;`
Opens and returns a file with name `f` and mode `m`. `m` may be one of `"r"` (for reading), `"w"` (for writing), `"r+w"` (for reading & writing), or `"a"` (for appending).

`f_close (f: File);`
Closes file `f`.

`f_getc (f: File): Char;`
Reads and returns the next character of file `f`.

`f_getl (f: File): Str;`
Reads and returns the next line of file `f`.

`f_zap (file: File);`
Reads and ignores to the end of the current line of file `f`.

`f_copy (f1: Str, f2: Str);`
Copies the contents of file `f1` to file `f2`.

`unix(com: Str): Int;`
Executes `com` as a UNIX command and returns the status as an integer.

`inp` - standard input.
`outp` - standard output.
`EOF` - end of file marker.

C.4 scr

`init_scr ();`

Initialises and clears the vdu screen and forces the terminal into special modes for screen io. The cursor is moved to the top left hand corner of the screen. This function must be called before any other function in the `scr` library.

`tini_scr ();`
Performs the reverse of `init_scr` by restoring the original modes of the terminal.

`clear ();`
Clears the vdu screen.

`move (lin: Nat, col: Nat);`
Moves the cursor to the coordinates `(lin,col)`. If this lies outside the screen then it will be automatically adjusted to the nearest position inside the screen.

`w_open (lins: Nat, cols: Nat, titl: Str);`
Opens a window with its origin positioned at the current position of the cursor. The window will be `lins` lines long and `cols` columns wide. The title `titl` will be displayed on top of the window. If the window, or part of it, lies outside the screen then its position will be automatically adjusted to the nearest suitable position. A window larger than the entire screen will be reduced to the size of the screen.

`w_close (n: Nat0);`
Closes the `n` most recently opened windows in the reverse order of openning. The cursor will be moved back to its original position, i.e. where it was before the window was opened.

`w_move (lin: Nat, col: Nat);`
Moves the cursor to the local coordinates `(lin,col)` inside the current window. If the position lies outside the window then it will be automatically adjusted to the nearest position inside the window.

`w_clear ();`
Clears the contents of the current window. The cursor will be moved to the top left hand corner of the window.

`w_scroll (n: Int);`
This function first awiats the press of a key (any key will do). It will then scroll the current window by `n` lines. A negative `n` specifies the number of lines of the old text to be kept after a scroll. If `n` is zero then the window will be scrolled `h-1` lines where `h` is the height of the window.

`w_text (lins: Nat, cols: Nat, titl: Str, tex: File | Str-list);`
This function first opens a window of the specified size and title (see `w_open`), and then displays `tex` in the window. `tex` may be a text file or a string list. The window may be scrolled as many times as necessary to accommodate the whole text. Once the entire text is displayed the window will be closed upon pressing any key.

`w_spec (spec: Char): Int;`
This function may be used to obtain the specification of current window according to the following values for `spec`:

- `'L'`: Length of window.
- `'C'`: Hight of window.
- `'l'`: Origin line of window.
- `'c'`: Origin column of window.

`bell ();`
Rings the margin bell.

`keybd ();`

Returns the next key stroke.

wait (n: *Nat0*);
Waits for n seconds.

time (t: *Char*): *Int*;
Returns the current time according to the following values for t:
'Y': Year
'M': Month
'D': Day
'h': Hour
'm': Minute
's': Second

fm_new (f: *Form* *, titl: *Str*);
Displays the form f in a window having the title titl. The user is then invited to fill the form interactively.

fm_view (f: *Form* *, titl: *Str*);
Displays the form f in a window having the title titl. The specification of the window is deduced from the form itself.

fm_drain (f: *Form* *);
Drains the image of the form f.

fm_put (file: *File*, f: *Form* *);
Writes the image of the form f to file.

fm_get (file: *File*, f: *Form* *);
Reads the image of the form f from file.

C.5 dbase

db_init (db: **-dbase*);
Initialises the databse db.

db_size (db: **-dbase*): *Nat0*;
Returns the size (i.e. the number of records) of db.

db_insert (db: **-dbase*, rec: **): *Bool*;
Inserts the record rec in database db provided it is not already there. A successful insertion will return TRUE; a failure will return FALSE.

db_delete (db: **-dbase*, k: ***): *Bool*;
Deletes the record whose key matches k from db provided it is already in the database. If successful it will return TRUE, otherwise it will return FALSE.

db_find (db: **-dbase*, k: ***): * | *NIL*;
Finds and returns the record in db whose key matches k. If no record with such key exists then NIL will be returned.

db_list (db: **-dbase*): **-list*;
Returns a list of records in db.

Appendix D THE LIBRARY SYSTEM

D.1 FUNCTIONAL SPECIFICATION

```
DOM Id     = Nat0;
    Code   = Nat0;
    Name   = Str;
    Date   = Nat0;
    Days   = Nat;
    Author = Str;
    Title  = Str;
    Volume = Nat0;
    Recall = Nat0;

ADT Lib
  DOM Lib     :: .rds:  Id   -> Reader, /* registered readers */
                 .stk:  Code -> Book,   /* library stock */
                 .loan: Code -> Loan,   /* current loans */
                 .top:  Top;            /* top indicators */

      Reader  :: .name: Name,           /* reader's name */
                 .join: Date,           /* joining date */
                 .leav: Date,           /* leaving date */
                 .loan: Code-set;       /* books borrowed */

      Book    :: .auth: Author,         /* author's name */
                 .titl: Title,          /* book title */
                 .vol:  Volume;         /* book volume no. */

      Loan    :: .date: Date,           /* date of loan/renew/discharge */
                 .rd:   [Id],           /* reader */
                 .res:  Reserve-list,   /* reservation list */
                 .rec:  Recall;         /* no. of recalls */

    Reserve :: .date: Date,             /* date of reservation */
               .rd:   Id,               /* reader */
               .till: [Date];           /* reserved until */

    Top     :: .code: Code,             /* last book code */
               .id:   Id,               /* last reader id. */
               .date: Date;             /* current date */

    Report  :: .lvs: Id-set,            /* leavers - with no loan */
               .dis: Id-set,            /* dishonoured readers */
               .rcs: Code-set,          /* recalled books */
               .rss: Code-set,          /* reserved books - now available */
               .lst: Code-set,          /* lost books */
               .rsf: Code -> Reserve-list;
                                        /* reserve failures due to loss */
TYPE
    del_id: Reserve-list, Id --> Reserve-list;
AUX
    inv-Lib((rds,stk,loan,top)) ==
        dom loan .S. dom stk &
        (.A id £ dom rds:
           let rd = rds(id) in
               (let ln = rd.loan in
                (.A cd £ ln: cd £ dom loan & loan(cd).rd = id)  &
                (let el = {cd: cd £ ln & top.date - loan(cd).date > 200} in
                     (rd.leav > top.date |
                      (ln /= {} & (.A cd £ ln: loan(cd).rec > 0)))          &
                     card ln <= 40 & (card ln = 0 | card ln > card el))))   &
```

```
        (.A cd £ dom loan:
            let (-,rd,rs,rc) = loan(cd) in
                (rd /= NIL | rs /= <> | rc > 0)                                    &
                (rd = NIL & rc = 0 & rs /= <> ==>
                          ((hd rs).till /= NIL & (hd rs).till > top.date)) &
                rc <= 4                                                            &
                (rd = NIL | (.E! rd £ rng rds: cd £ rd.loan) &
                            cd £ rds(rd).loan)                                     &
                (let rss = elems rs in
                     (.A rz £ rss: rz.rd /= rd &
                                   (rz.till = NIL | top.date > rz.till))));

    del_id(rs,id) == mac {
                       rs = <>          => <>,
                       (hd rs).rd = id  => tl rs,
                       TRUE             => <hd rs> || del_id(tl rs,id),
                      };
OPS
    /* initialise the library */
    INIT: --> ;
      post(-,lib') == lib' = mk-Lib([],[],[],mk-Top(0,0,0));
    END INIT

    /* register a new reader */
    NEW_READ: Name, Days --> Id;
      post((rds,stk,loan,top),name,days,lib',id) == lib' =
          mk-Lib(rds+
                 [top.id+1 -> mk-Reader(name,top.date,top.date+days,{})],
                 stk,loan,
                 mk-Top(top.code,top.id+1,top.date)) &
          id = top.id+1;
    END NEW_READ
```

```
    /* de-register a reader */
    REM_READ: Id --> Code-set;
      exep((rds,-,loan,-),id) ==
          ~(id £ dom rds)              => "No such reader",
          rds(id).loan /= {}           => "Has still books on loan";
      post((rds,stk,loan,top),id,lib',cs) ==
          (let ln = [cd -> l: cd £ dom loan &
                             (let (dt,rd,rs,rc) = loan(cd) in
                                  l = mk-Loan(dt,rd,del_id(rs,id),rc))] in
               cs = {cd: cd £ dom ln & (let (-,rd,rs,rc) = ln(cd) in
                                            rd = NIL & rs = <> & rc = 0)} &
               lib' = mk-Lib(rds /- {id},stk,ln /- cs,top));
    END REM_READ

    /* add a new book to the library */
    NEW_BOOK: Author, Title, Volume --> Code;
      post((rds,stk,loan,top),auth,titl,vol,lib',code) ==
          lib' = mk-Lib(rds,
                        stk + [top.code+1 -> mk-Item(auth,titl,vol)],
                        loan,
                        mk-Top(top.code+1,top.id,top.date)) &
          code = top.code+1;
    END NEW_BOOK

    /* remove a book from the library */
    REM_BOOK: Code --> Reserve-list;
      exep(lib,code) ==
          ~(code £ dom lib.stk) => "No such book";
      post((rds,stk,loan,top),code,lib',rsv) ==
          rsv = loan(code).res &
          lib' = mk-Lib(rds,stk /- {code},loan /- {code},top);
```

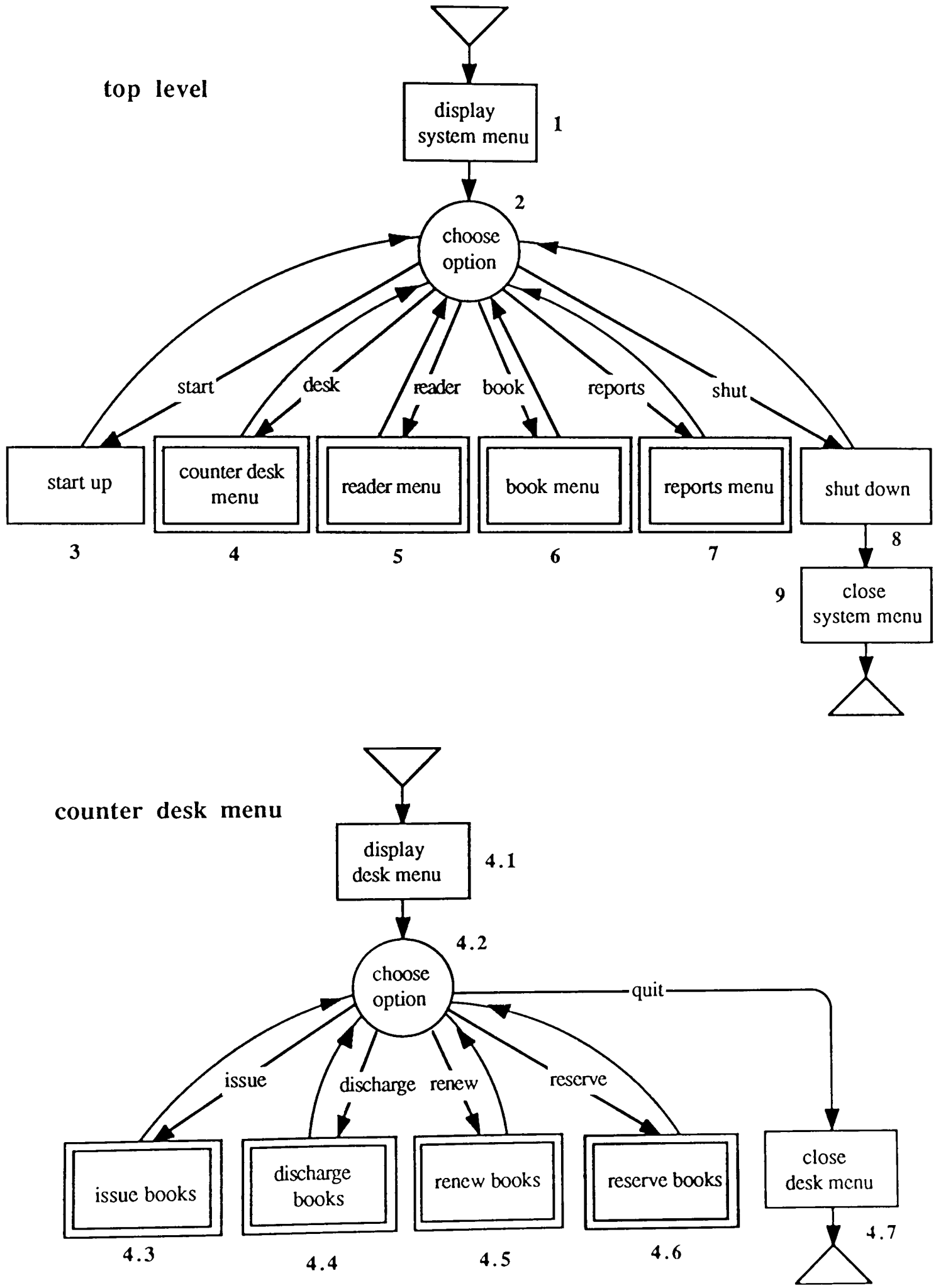
```
END REM_BOOK

/* issue a book for a reader */
ISSUE: Id, Code --> ;
  exep((rds,stk,loan,top),id,code) ==
      ~(id £ dom rds)              => "No such reader",
      ~(code £ dom stk)            => "No such book",
      code £ dom loan &
      (let ln = loan(code) in
           ln.rd /= NIL | ln.rec /= 0 |
           (ln.res /= <> & (hd ln.res).rd /= id))
                                   => "Already on loan",
      rds(id).leav < top.date      => "Reader's Reg. expired",
      card {cd: cd £ rds(id).loan & loan(cd).rd = id} >= 40
                                   => "Borrow limit reached";

  post((rds,stk,loan,top),id,code,lib') == lib' =
      mk-Lib(rds ++ [id -> let (nm,jn,lv,ln) = rds(id) in
                           mk-Reader(nm,jn,lv,ln .U. {code})],
             stk,
             loan ++ [code -> mk-Loan(top.date,id,
                                      if code £ dom loan then
                                         tl loan(code).res
                                      else <>,
                                      0)],
             top);
END ISSUE

/* discharge a book */
DISCHARGE: Code --> [Id];
  exep(lib,code) ==
      ~(code £ dom lib.stk)  => "No such book",
      ~(code £ dom lib.loan) => "Is not on loan";

  post((rds,stk,loan,top),code,lib',id) ==
      let (-,rd,rs,rc) = loan(code) in
      let ln = mk-Loan(top.date,NIL,
                       if rs /= <> & rc = 0 then
                          let (dt,rd,-) = hd rs in
                              <mk-Reserve(dt,rd,top.date+14)> || tl rs
                       else rs,
                       rc) in
          lib' = mk-Lib(rds ++ [loan(code).rd ->
                                let (nm,jn,lv,ln)=rds(loan(code).rd) in
                                    mk-Reader(nm,jn,lv,ln - {code})],
                        stk,loan ++ [code -> ln],top) &
          id = (if rs /= <> & rc = 0 then (hd rs).rd
                else NIL);
END DISCHARGE

/* renew a book */
RENEW: Code --> ;
  exep((-,stk,loan,top),code) ==
      ~(code £ dom stk)     => "No such book",
      ~(code £ dom loan)    => "Is not on loan",
      loan(code).rec /= 0   => "Recalled - can't renew",
      loan(code).res /= <>  => "Reserved - can't renew";

  post((rds,stk,loan,top),code,lib') == lib' =
      mk-Lib(rds,stk,
             loan ++ [code -> mk-Loan(top.date,loan(code).rd,<>,0)],
             top);
END RENEW
```

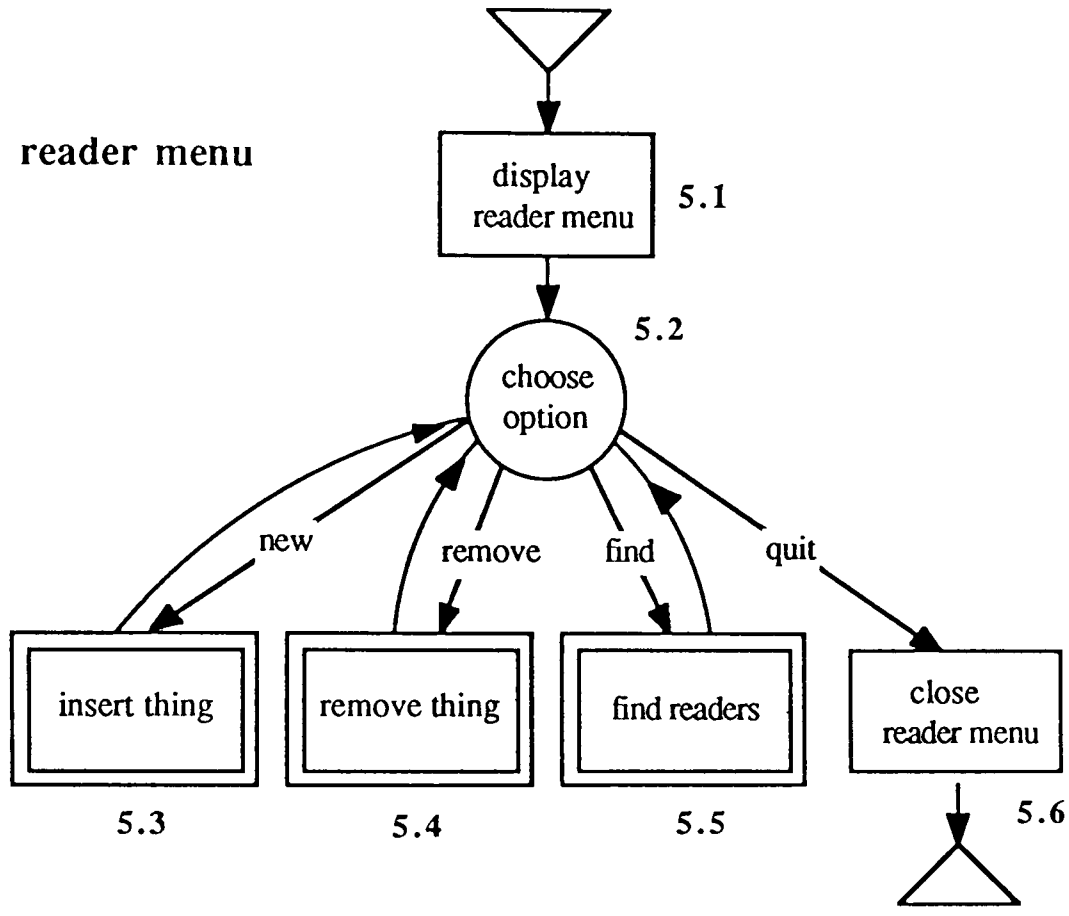
```
/* reserve an book */
RESERVE: Id, Code --> ;
  exep((-,stk,loan,-),id,code) ==
      ~(code £ dom stk)    => "No such book",
      ~(code £ dom loan)   => "Is not on loan",
      loan(code).rd = id   => "You have the book - can't reserve",
      loan(code).res /= <> &
      (.E rs £ elems loan(code).res: rs.rd = id)
                           => "Already reserved for you";

  post((rds,stk,loan,top),id,code,lib') ==
      let (dt,rd,rs,rc) = loan(code) in
          lib' = mk-Lib(rds,stk,
                        loan ++ [code ->
                                 mk-Loan(dt,rd,
                                         rs || <mk-Reserve(top.date,id,NIL)>,
                                         rc)],
                        top);
END RESERVE

/* check a recalled book which has been returned */
CHECKED: Code --> ;
  exep((-,-,loan,-),code) ==
      ~(code £ dom loan) => "Is not on loan",
      loan(code).rec = 0 => "Was not recalled";
  post((rds,stk,loan,top),code,lib') ==
      lib' = mk-Lib(rds,stk,
                    loan ++ [code -> let (dt,rd,rs,-) = loan(code) in
                                     mk-Loan(dt,rd,rs,0)],
                    top);
END CHECKED
```

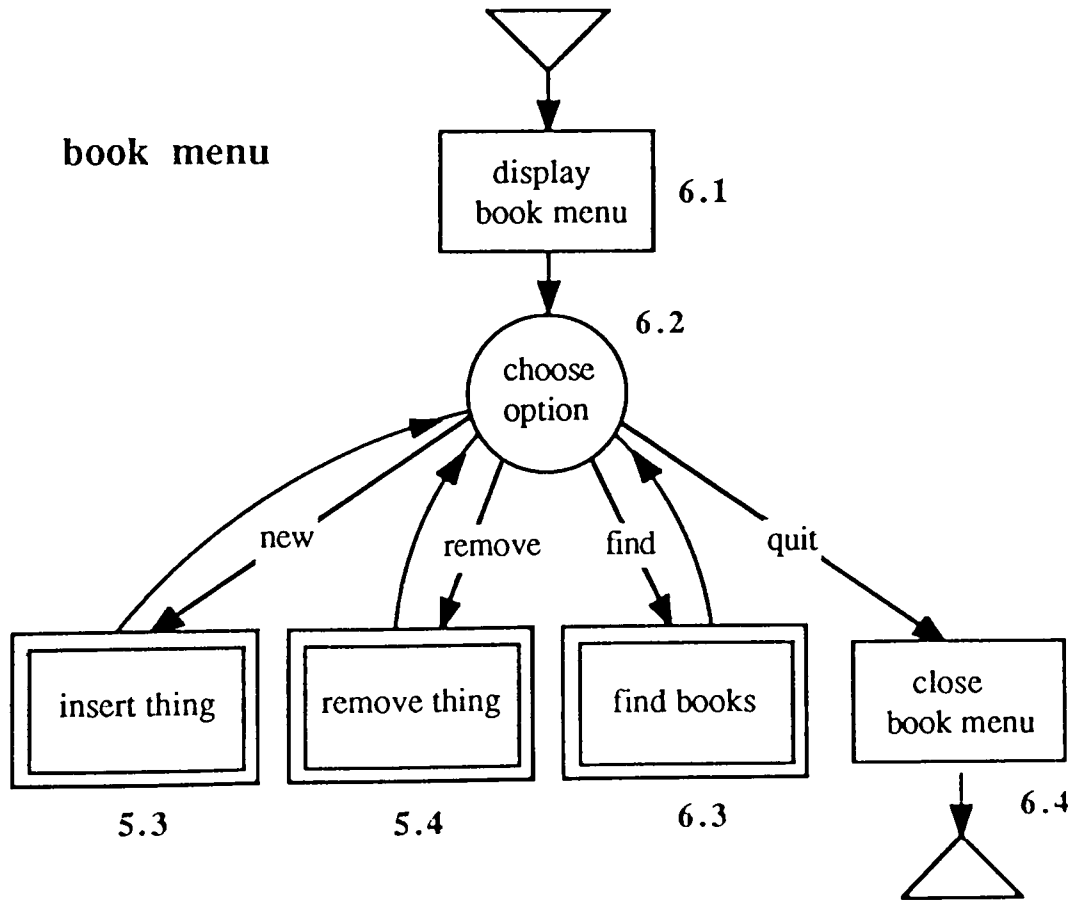
```
/* daily operation - to be performed once a day */
DAILY: --> Report;
  post((rds,stk,loan,top),lib',rep) ==
      let ex  = {id: id £ dom rds & rds(id).leav < top.date},
          lon = [cd -> ln: cd £ dom loan &
                           ln = (let (dt,rd,rs,rc) = loan(cd) in
                                     if rd = NIL & rs /= <> & rc = 0 &
                                        (hd rs).till /= NIL &
                                        (hd rs).till < top.date then
                                        mk-Loan(dt,rd,tl rs,rc)
                                     else loan(cd))],
          ls  = {cd: cd £ dom loan & top.date - loan(cd).date > 200} in
      let dis = {id: id £ dom rds & rds(id).loan /= {} &
                     rds(id).loan .S. ls} in

      let lvs = {id: id £ ex & rds(id).loan = {}},
          rcs = {cd: cd £ dom lon &
                     (let (dt,rd,rs,rc) = lon(cd) in
                          rd /= NIL & rc < 4 &
                          lon(cd).rec*30+14 <= top.date - dt)} .U.
                (union {rds(id).loan: id £ ex} -
                 {cd: cd £ dom lon & lon(cd).rec > 0}),
          rss = {cd: cd £ dom lon & (let (-,rd,rs,rc) = lon(cd) in
                                         rd = NIL & rs /= <> &
                                         rc = 0 & (hd rs).till = NIL)},
          lst = {cd: cd £ ls & lon(cd).rd £ dis} in
      let rsf = [cd -> lon(cd).res: cd £ lst & lon(cd).res /= <>] in
          rep = mk-Report(lvs,dis,rcs,rss,lst,rsf) &
          lib' = mk-Lib(rds /- (lvs .U. dis),
                        stk /- lst,
                        [cd -> ln:
                         cd £ (dom lon - lst) &
```

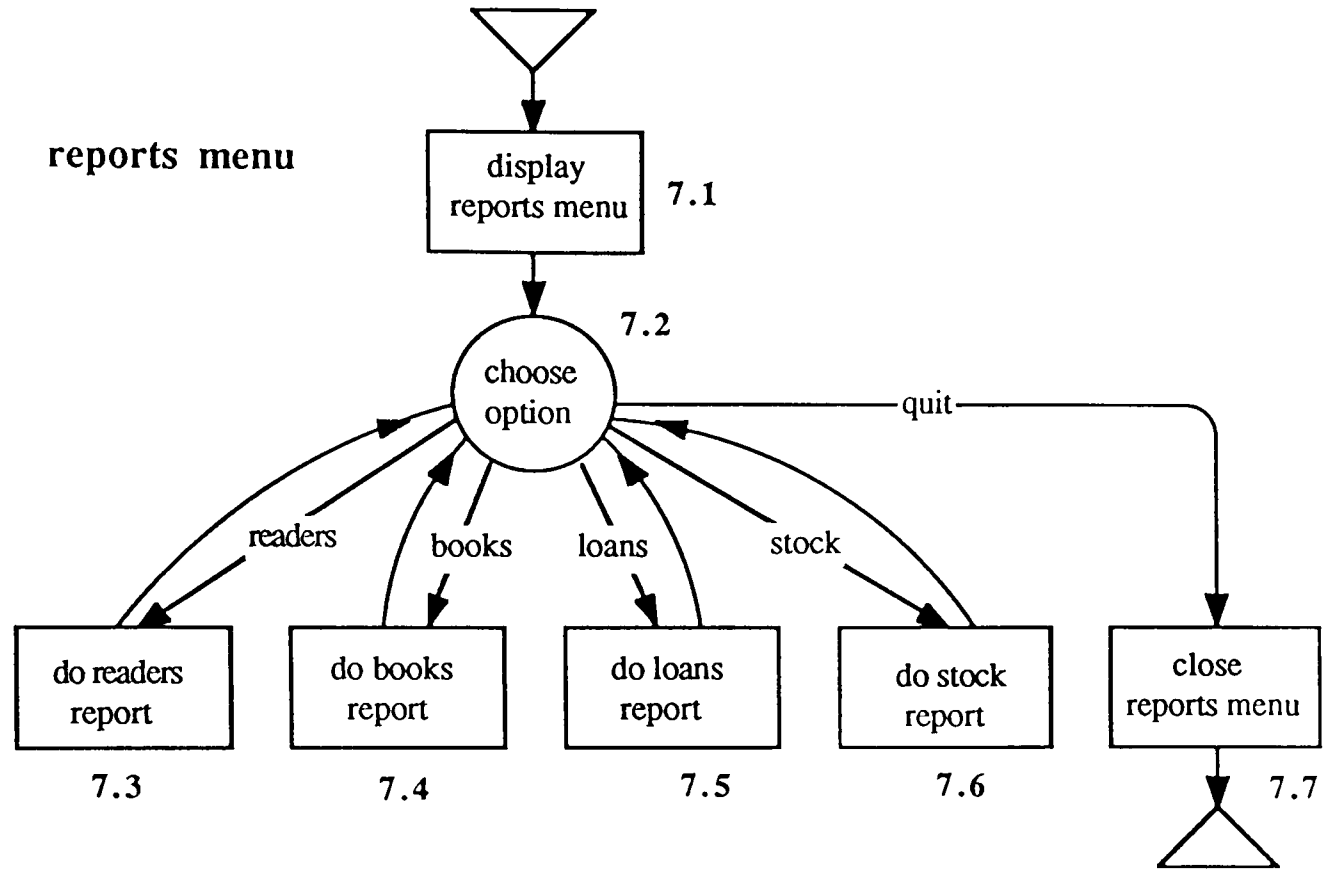
```
                        (let (dt,rd,rs,rc) = lon(cd) in
                         ln = mk-Loan(dt,rd,
                                      if cd £ rss then
                                         <mk-Reserve((hd rs).date,
                                                     (hd rs).rd,
                                                     top.date+14)> || tl rs
                                      else rs,
                                      if cd £ rcs then rc+1 else rc))],
                        mk-Top(top.code,top.id,top.date+1));
    END DAILY
END Lib
```

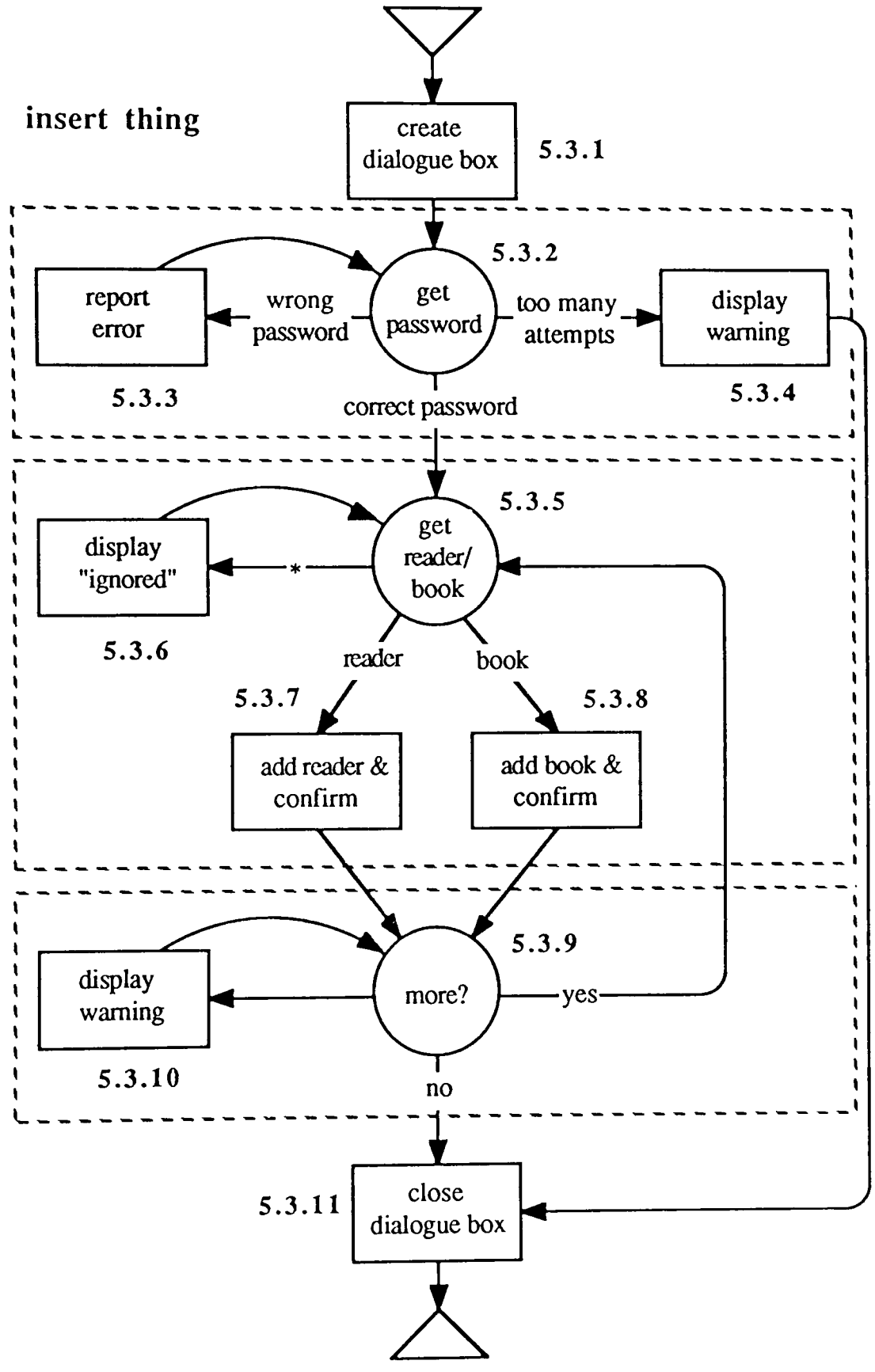
D.2 USER INTERFACE SPECIFICATION

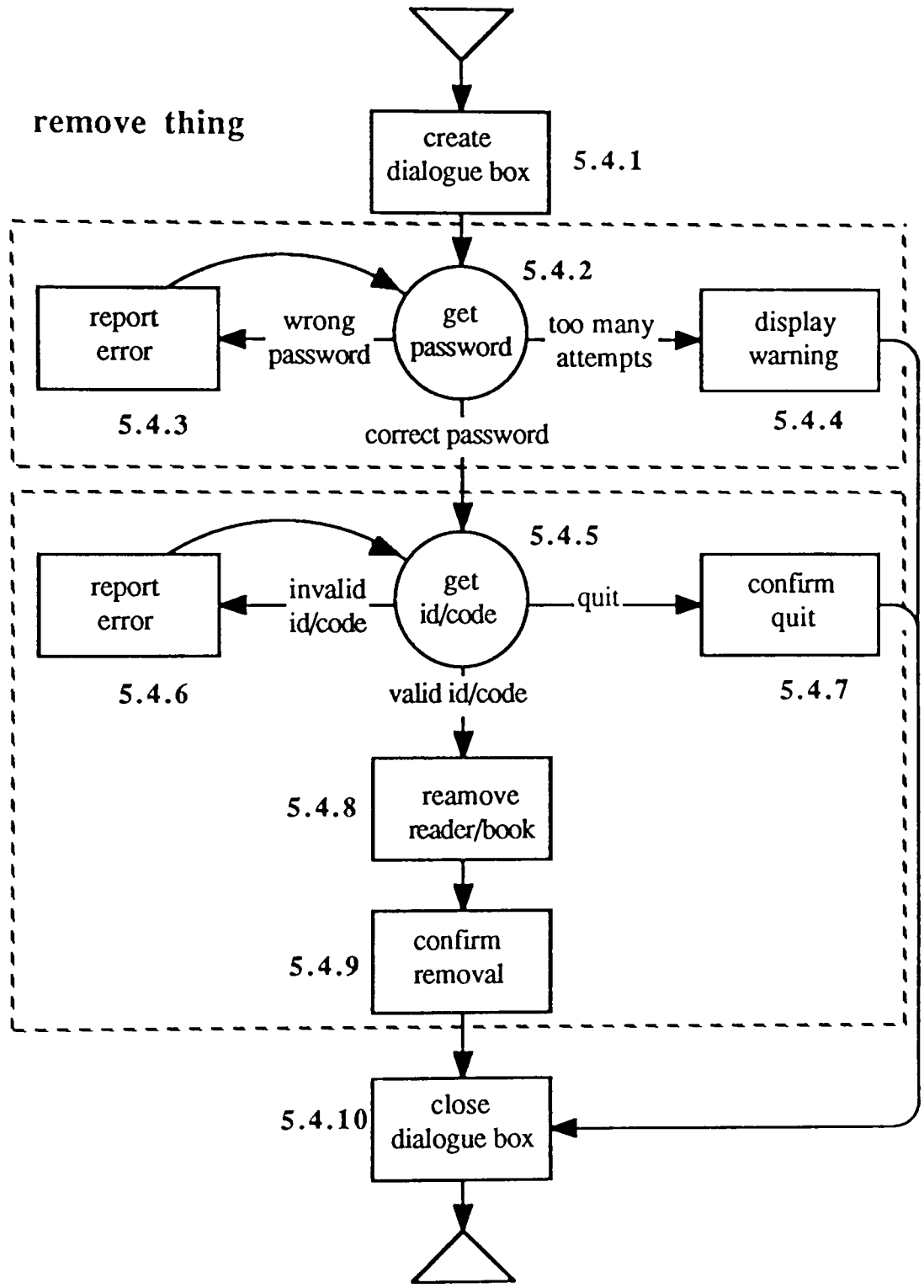


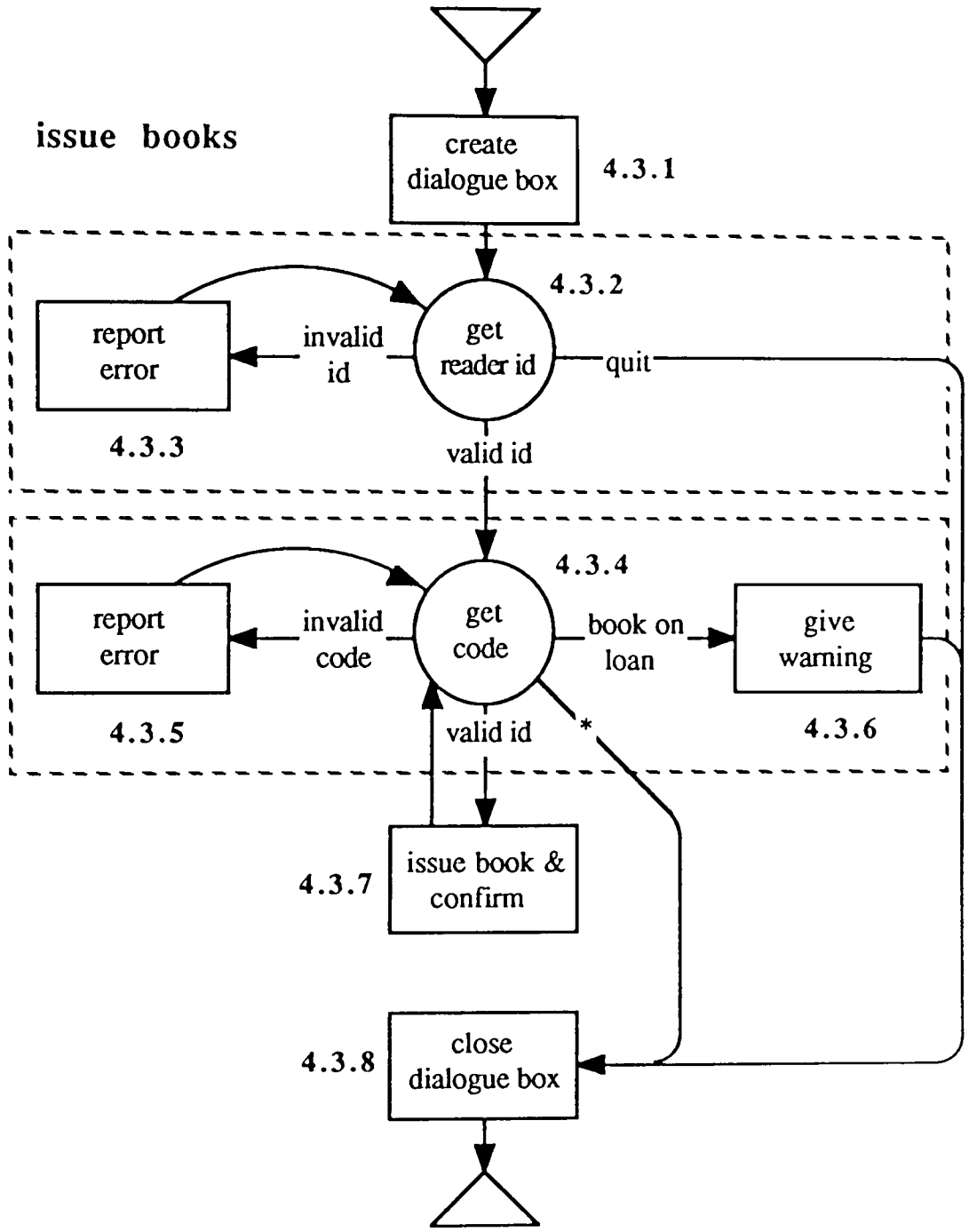

reader menu
display reader menu
5.1
5.2
choose option
new
remove
find
quit
insert thing
remove thing
find readers
close reader menu
5.3
5.4
5.5
5.6

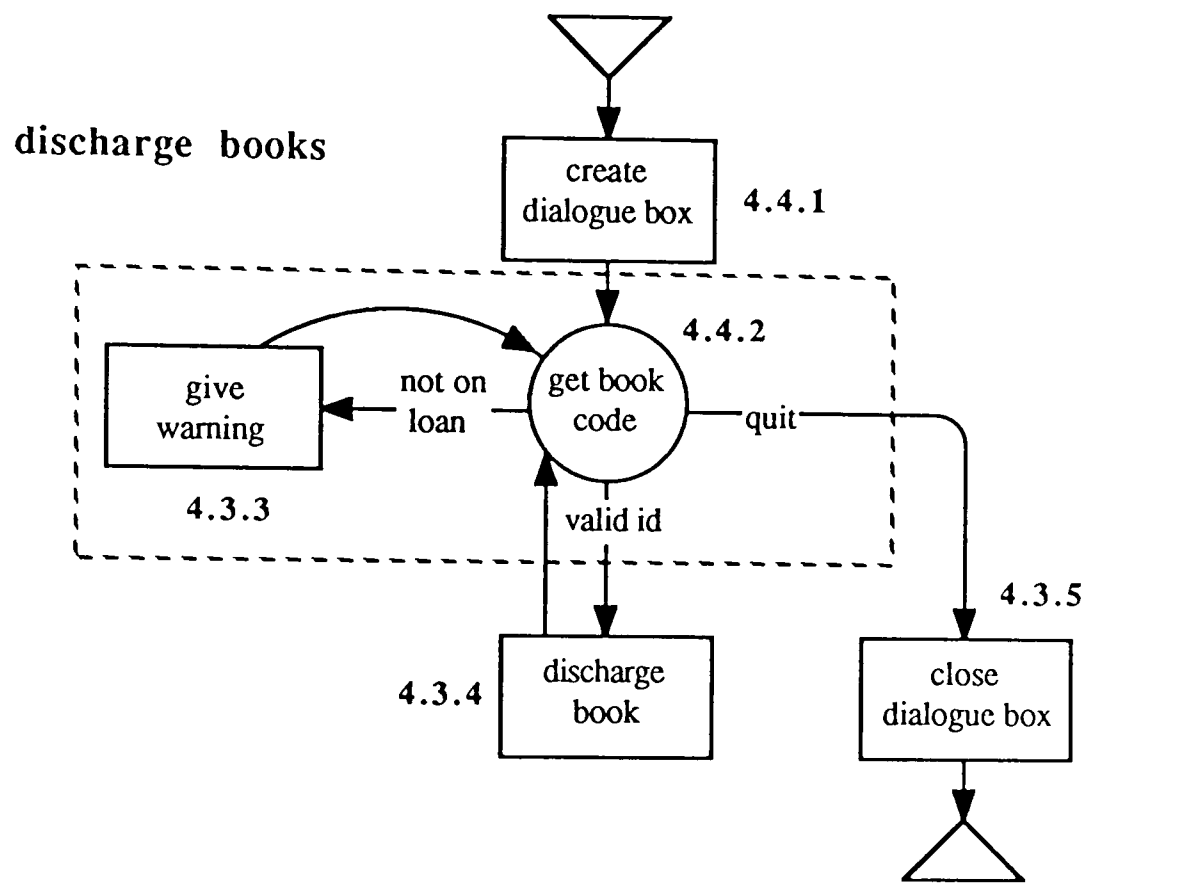

book menu
display book menu
6.1
6.2
choose option
new
remove
find
quit
insert thing
remove thing
find books
close book menu
5.3
5.4
6.3
6.4

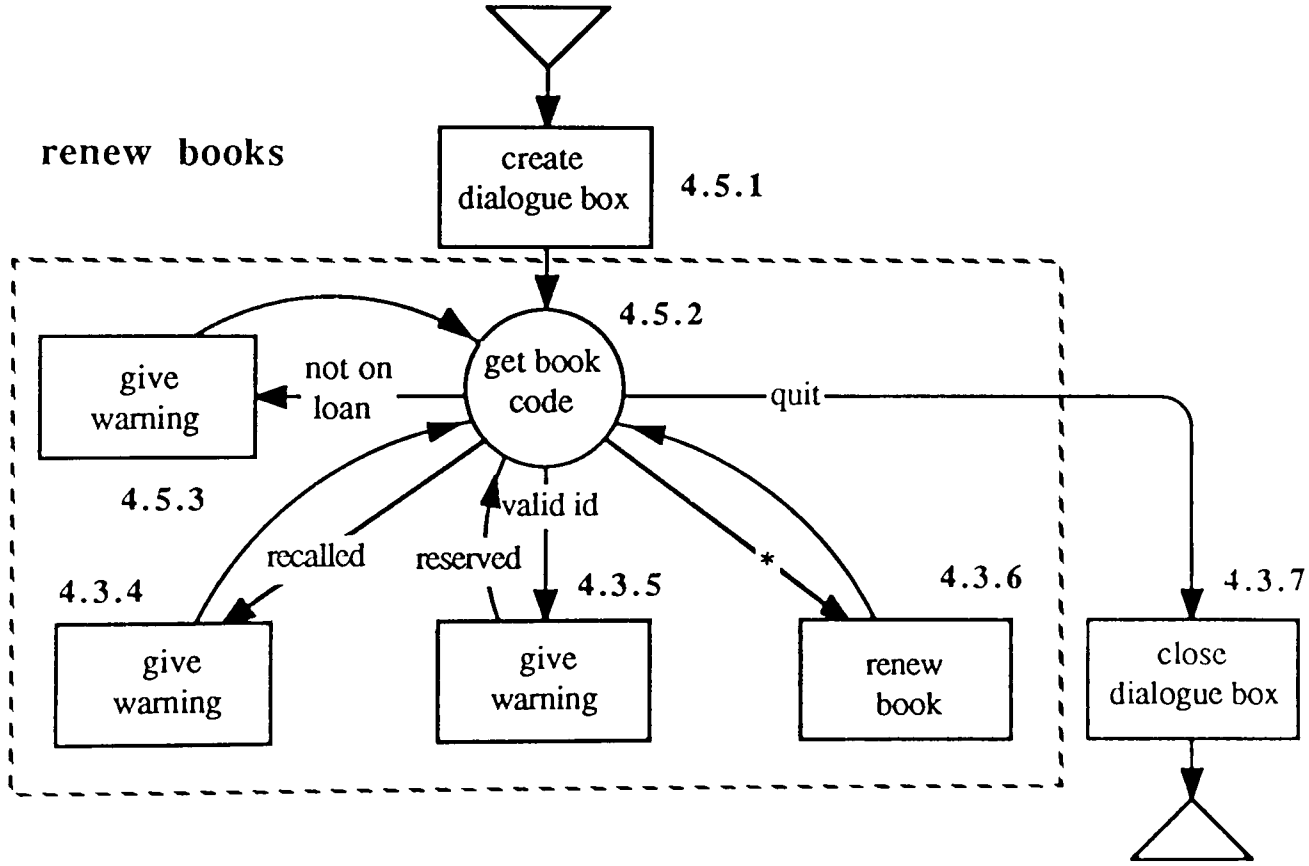

reports menu
display reports menu
7.1
7.2
choose option
quit
readers
books
loans
stock
do readers report
do books report
do loans report
do stock report
close reports menu
7.3
7.4
7.5
7.6
7.7

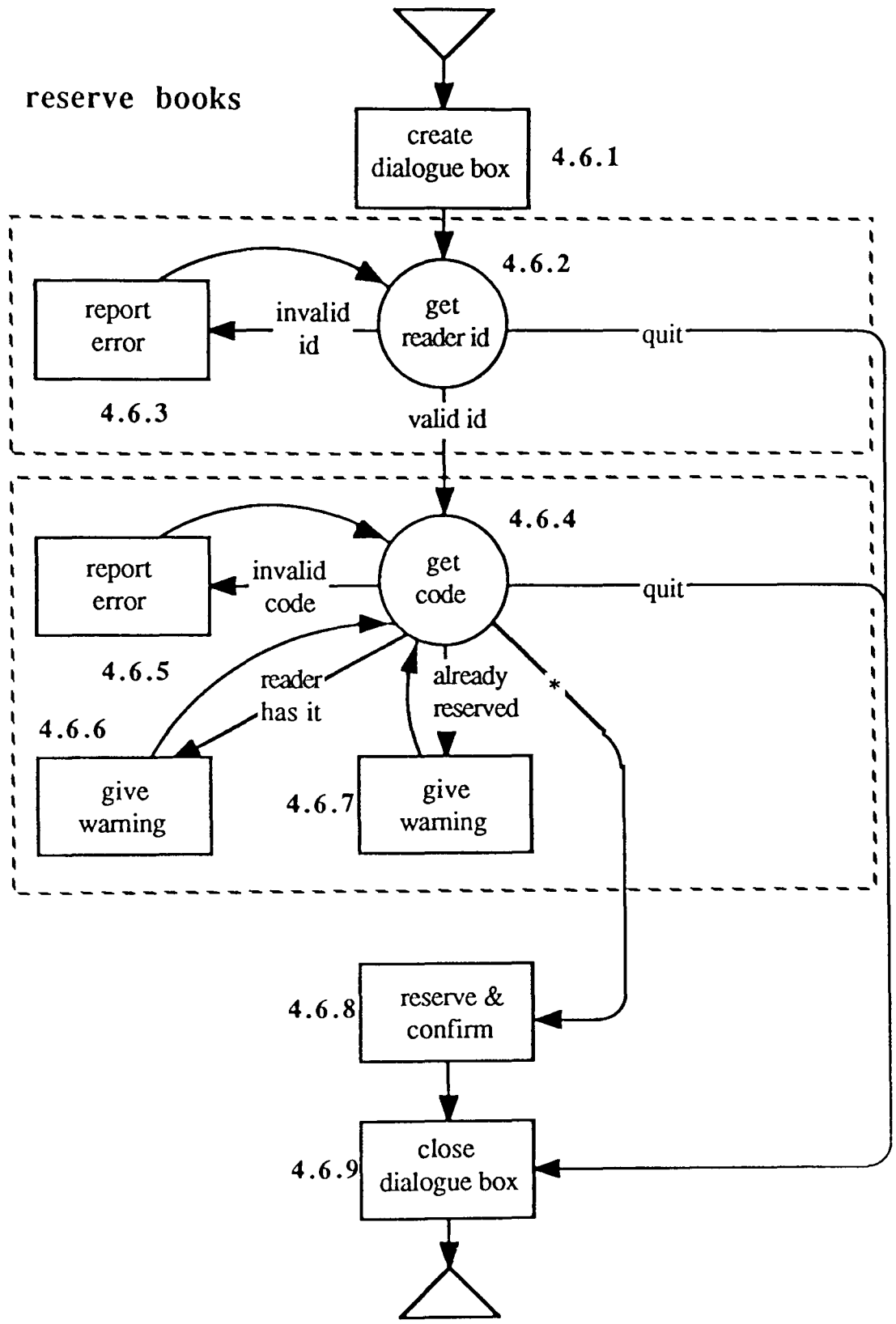

insert thing
create dialogue box
5.3.1
5.3.2
report error
wrong password
get password
too many attempts
display warning
5.3.3
correct password
5.3.4
5.3.5
display "ignored"
*
get reader/ book
5.3.6
reader
book
5.3.7
5.3.8
add reader & confirm
add book & confirm
5.3.9
display warning
more?
yes
5.3.10
no
5.3.11
close dialogue box


remove thing
create dialogue box
5.4.1
5.4.2
report error
wrong password
get password
too many attempts
display warning
5.4.3
correct password
5.4.4
5.4.5
report error
invalid id/code
get id/code
quit
confirm quit
5.4.6
valid id/code
5.4.7
5.4.8
reamove reader/book
5.4.9
confirm removal
5.4.10
close dialogue box

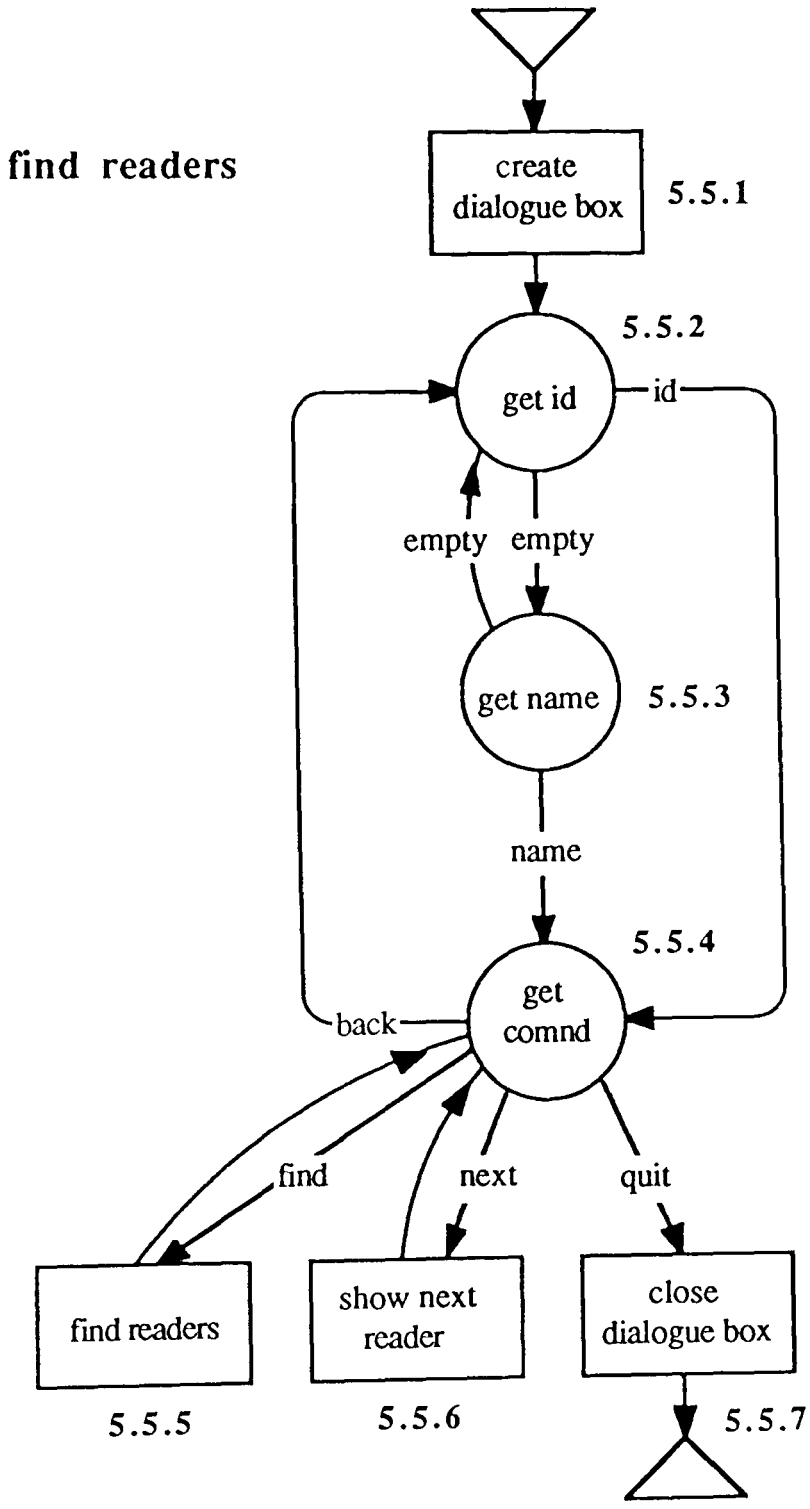

issue books
create dialogue box
4.3.1
4.3.2
report error
invalid id
get reader id
quit
4.3.3
valid id
4.3.4
report error
invalid code
get code
book on loan
give warning
4.3.5
valid id
*
4.3.6
4.3.7
issue book & confirm
4.3.8
close dialogue box

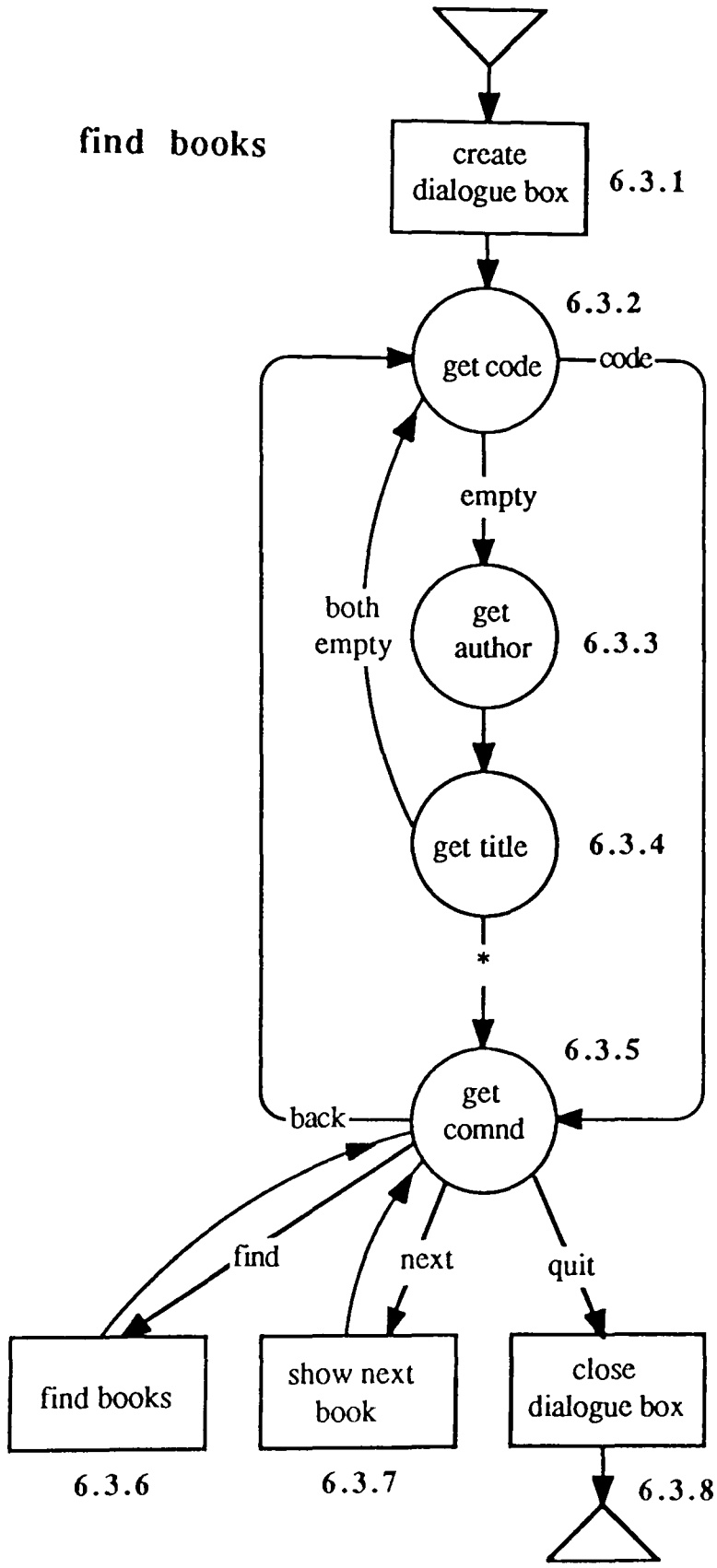

discharge books
create dialogue box
4.4.1
4.4.2
give warning
not on loan
get book code
quit
4.3.3
valid id
4.3.5
4.3.4
discharge book
close dialogue box


renew books
create dialogue box
4.5.1
4.5.2
give warning
not on loan
get book code
quit
4.5.3
valid id
recalled
reserved
*
4.3.4
4.3.5
4.3.6
4.3.7
give warning
give warning
renew book
close dialogue box


reserve books
create dialogue box
4.6.1
4.6.2
get reader id
invalid id
report error
4.6.3
quit
valid id
4.6.4
get code
invalid code
report error
4.6.5
quit
reader has it
already reserved
*
4.6.6
give warning
4.6.7
give warning
4.6.8
reserve & confirm
4.6.9
close dialogue box

find readers




find books
create dialogue box
6.3.1
6.3.2
get code
code
empty
both empty
get author
6.3.3
get title
6.3.4
*
6.3.5
get comnd
back
find
next
quit
find books
6.3.6
show next book
6.3.7
close dialogue box
6.3.8

D.3 FINAL PROTOTYPE

```
%library "scr"          /* screen management library */
%library "str"          /* string library */
%library "io"           /* IO library */
%library "dbase"        /* database library */

CONST MONTHS = <"Jan","Feb","Mar","Apr","May","Jun",
                "Jul","Aug","Sep","Oct","Nov","Dec">;
      DEL_PASS    = "r2d2";
      INS_PASS    = "x2y2";
      ATTEMPT_LIM = 3;
      READER = 'R';
      BOOK   = 'B';
      NOTE   = 'N';
      WARN   = 'W';

DOM Id       = Nat0;
    Code     = Nat0;
    Author   = Str;
    Title    = Str;
    Date     = Nat;
    Day_no   = Nat;
    Recall   = {0:4};
    Name     = Str;
    Position = Nat0;
    What     = {READER,BOOK};
    Message  = {NOTE,WARN};

DESIGN library_system();
DOM Reader :: .id:    Id,
              .pos:   Position,
              .name:  Name,
              .valid: Bool,
              .count: Nat0,
              .loan:  Code-list;

    Book ::   .code:  Code,
              .pos:   Position,
              .auth:  Author,
              .titl:  Title;

    Loan ::   .code: Code,
              .rd:   [Id],
              .date: Date,
              .rec:  Recall,
              .res:  Reserve-list;

    Reserve :: .date: Date,
               .rd:   Id,
               .till: [Date];

    TopDate :: .no: Nat, .y:  Nat, .m: Nat, .d: Nat;

    ReadersDb = Reader-dbase(key = id);
    BooksDb   = Book-dbase(key = code);
    LoansDb   = Loan-dbase(key = code);
    ReaderForm = form ReaderReg;
    BookForm   = form BookRec;
```

```
VAR rds_db: ReadersDb;
    bks_db: BooksDb;
    lns_db: LoansDb;

    rds_list: Reader-list := <>;
    bks_list: Book-list   := <>;

    new_rds:  ReaderForm-list := <>;
    rmv_rds:  Id-list         := <>;
    new_bks:  BookForm-list   := <>;
    rmv_bks:  Bool := TRUE;
    ins_ok:   Bool := TRUE;
    del_ok:   Bool := TRUE;
    started:  Bool := FALSE;
    stock_rep_ready: Bool := FALSE;
    day_no:   Nat;
    cur_date: TopDate;

FORM ReaderReg
   \
                                        Date:  $date
     Surname:   $sname                  Title: $title
     Forenames: $fname1           $fname2
     Position:  $pos
     Faculty:   $fac              Extension: $ext
     Home Address: $road
                   $town
                   $pcode
  Telephone No: $telno       Leaving date: $d/$m/$y \

$date:   Str(8), system(sdate);
$sname:  Str(20);
$title:  Str(4);
$fname1: Str(15);
$fname2: Str(15), optional;
$pos:    Str(2), computed menu {
                        "^M Position ^N"
                        "Dean"                => ($pos := "DN"; exit);
                        "Senior Lecturer"     => ($pos := "SL"; exit);
                        "Lecturer"            => ($pos := "LC"; exit);
                        "Visitor"             => ($pos := "VS"; exit);
                        "Research Fellow"     => ($pos := "RF"; exit);
                        "Reasearch Assistant" => ($pos := "RA"; exit);
                        "Research Student"    => ($pos := "RS"; exit);
                        "Technician"          => ($pos := "TC"; exit);
                        "Secretary"           => ($pos := "SC"; exit);
                       };
$fac:    Str(11), computed menu {
                         "^M Faculty ^N"
                         "Art"          => ($fac := itself; exit);
                         "Education"    => ($fac := itself; exit);
                         "Geography"    => ($fac := itself; exit);
                         "Mathematics"  => ($fac := itself; exit);
                         "Sciences"     => ($fac := itself; exit);
                         "Technology"   => ($fac := itself; exit);
                        };
$ext:    Nat(4), constraint 1000 <= $ext <= 9999;
$road:   Str(30);
```

```
    $town:   Str(30);
    $pcode:  Str(7), optional;
    $telno:  Nat(7), optional;
    $d:      Nat(2), constraint 1 <= $d <= 31;
    $m:      Nat(2), constraint 1 <= $m <= 12;
    $y:      Nat(2), constraint time('Y') <= $y <= 99;
END ReaderReg

FORM BookRec
  \
                                                    Date: $edate
     Class:  L$cl.$cr                      Purchase Date: $d/$m/$y
     Author: $i. $auth                             Year: $year
     Title:  $titl
     Volume: $vol                               Edition: $edtn
     Publsh: $pub                                  ISBN: 0-$s1-$s2  -$s3\

  $edate: Str(8),  system(sdate);
  $cl:    Nat0(3), constraint 0 <= $cl <= 799;
  $cr:    Nat0(3);
  $d:     Nat(2),  constraint 1 <= $d <= 31, initially time('D');
  $m:     Nat(2),  constraint 1 <= $m <= 12, initially time('M');
  $y:     Nat(2),  initially time('Y');
  $i:     Str(2);
  $auth:  Str(20);
  $year:  Nat(4),  after($y), constraint $year <= 1900 + $y;
  $titl:  Str(60);
  $vol:   Nat0(2), initially 0;
  $edtn:  Nat(2),  initially 1;
  $pub:   Str(30), optional;
  $s1:    Nat(3);
  $s2:    Nat(5);
  $s3:    Nat(1);
END BookRec

FUNCTION init_readers();
VAR rdf:      File;
    rds_cnt:  Nat0;
    id:       Id;
    pos:      Position := 0;
    valid:    Bool;
    count:    Nat0;
    loan:     Code;
    loans:    Code-list := <>;
    rd_fm:    ReaderForm;
BEGIN
    rdf := f_open("readers","r");
    db_init(rds_db);
    get(rdf,rds_cnt);
    f_zap(rdf);

    for i in (1:rds_cnt) do (
        get(rdf,id,valid,count);
        for j in (1: count) do (
            get(rdf,loan);
            loans := loans || <loan>;
        );
        fm_get(rdf,rd_fm);
        db_insert(rds_db,mk-Reader(id,pos,rd_fm.$sname,valid,count,loans));
```

```
        pos := pos+1;
    };
    f_close(rdf);
END init_readers

FUNCTION init_books();
VAR bkf:     File;
    bks_cnt: Nat0;
    code:    Code;
    bk_fm:   BookForm;
    pos:     Position := 0;
BEGIN
    bkf := f_open("books","r");
    db_init(bks_db);
    get(bkf,bks_cnt);
    f_zap(bkf);

    for i in {1:bks_cnt} do {
        get(bkf,code);
        fm_get(bkf,bk_fm);
        db_insert(bks_db, mk-Book(code,pos,bk_fm.$auth,bk_fm.$titl));
        pos := pos+1;
    };
    f_close(bkf);
END init_books

FUNCTION init_loans (): Day_no;
VAR lnf:     File;
    lns_cnt,rs_cnt: Nat0;
    date,rs_date:   Date;
    rd,rs_rd:       Id;
    code:    Code;
    rec:     Recall;
    rs_till: [Date];
    res:     Reserve-list := <>;
    day_no:  Nat;
BEGIN
    lnf := f_open("loans","r");
    db_init(lns_db);
    get(lnf,lns_cnt,day_no);
    f_zap(lnf);

    for i in {1:lns_cnt} do {
        get(lnf,code,rd,date,rec,rs_cnt);
        for j in {1:rs_cnt} do {
            get(lnf,rs_date,rs_rd,rs_till);
            res := res || <mk-Reserve(rs_date,rs_rd,
                                      if rs_till = 0 then NIL else rs_till)>;
        };
        db_insert(lns_db,mk-Loan(code,if rd=0 then NIL else rd,date,rec,res));
    };
    f_close(lnf);
    return(day_no+1);
END init_loans

FUNCTION is_element_of(obj: Id | Code, objl: (Id | Code)-list): Bool;
BEGIN
    while objl /= <> do {
          if obj = hd objl then
```

```
            return(TRUE);
          objl := tl objl;
    };
    return(FALSE);
END is_element_of

FUNCTION is_expired(y: Nat, m: Nat, d: Nat): Bool;
BEGIN
    return(y*365+m*30+d < cur_date.no);
END is_expired

FUNCTION update_readers(rds_db: ReadersDb, new_rds: ReaderForm-list, rmv_rds: Id-list);
VAR rdf,logf,tempf:    File;
    valid, stays:      Bool := FALSE;
    rds_cnt, lns_cnt:  Nat0;
    rd_fm:    ReaderForm;
    id :      Id   := 0;
    rds_cnt': Nat0 := 0;
    code:     Code;
    rd:       Reader;
    loans:    Code-list;
    loan:     [Loan];
BEGIN
    rdf := f_open("readers","r");
    logf := f_open("readers.log","w");
    tempf := f_open("temp","w");
    get(rdf,rds_cnt);
    f_zap(rdf);
    put(tempf,"%05d^n",rds_cnt');
    for i in {1:rds_cnt} do {
        get(rdf,id,valid,lns_cnt);
        for j in {1:lns_cnt} do
            get(rdf,code);
        fm_get(rdf,rd_fm);
        rd := db_find(rds_db,id);
        loans := rd.loan;

        if is_element_of(id,rmv_rds) then {
           put(logf,"* Reader Removed: %5d %s Reg. on %02d-%02d-%02d ^n",
                     id, rd_fm.$sname, rd_fm.$d, rd_fm.$m, rd_fm.$y);
           while loans /= <> do {
                 loan := db_find(lns_db,hd loans);
                 put(logf,"Lost Book: %06d by %05d ^n",loan.code,id);
                 loans := tl loans;
           }
        }
        else if is_expired(rd_fm.$y,rd_fm.$m,rd_fm.$d) then {
           if (rd.count > 0) then {
              rd.valid := FALSE;
              while loans /= <> do {
                  loan := db_find(lns_db,hd loans);
                  if loan.rec = 0 then {
                     loan.rec := 1;
                     put(logf,"Recall Book: %06d from %05d ^n",loan.code, id);
                  };
                  loans := tl loans;
              };
              stays := TRUE;
```

```
            }
            else
               put(logf,"Reader Removed: %05d %s Reg. on %02d-%02d-%02d ^n",
                         id, rd_fm.$sname, rd_fm.$d, rd_fm.$m, rd_fm.$y);
         }
         else
            stays := TRUE;
         if stays then {
            put(tempf,"%d %s %d",id,
                      if rd.valid then "TRUE" else "FALSE",rd.count);
            loans := rd.loan;
            for i in {1:rd.count} do {
                put(tempf, "%d", hd loans);
                loans := tl loans;
            };
            fm_put(tempf,rd_fm);
            rds_cnt' := rds_cnt'+1;
         };
      };
      while new_rds /= <> do {
            id := id + 1;
            rd_fm := hd new_rds;
            put(tempf,"%d %s %d",id,"TRUE",0);
            fm_put(tempf,rd_fm);
            put(logf,"New Reader: %5d %20s on %d-%d-%d^n",
                     id, rd_fm.$sname, rd_fm.$d, rd_fm.$m, rd_fm.$d);
            rds_cnt' := rds_cnt'+1;
            new_rds := tl new_rds;
      };
      f_close(rdf);
      f_close(tempf);
      f_close(logf);
      tempf := f_open("temp","r+w");
      put(tempf,"%05d^n",rds_cnt');
      f_close(tempf);
      f_copy("temp","readers");
   END update_readers

   FUNCTION update_books(new_bks: BookForm-list, rmv_bks: Code-list);
   VAR bkf,logf,tempf: File;
       bk_fm:    BookForm;
       id:       Id;
       code :    Code := 0;
       pos  :    Position;
       stays:    Bool := FALSE;
       bks_cnt:  Int;
       bks_cnt': Int := 0;
       bk:       Book;
   BEGIN
       bkf := f_open("books","r");
       logf := f_open("books.log","w");
       tempf := f_open("temp","w");
       get(bkf,bks_cnt);
       f_zap(bkf);
       put(tempf,"%05d^n",bks_cnt');

       for i in {1:bks_cnt} do {
           get(bkf,code);
           fm_get(bkf,bk_fm);
```

```
        if is_element_of(id,rmv_bks) then
           put(logf,"Book Removed: %06d %s-%4d Purch. on %02d-%02d-%02d ^n",
                     code,bk_fm.$auth,bk_fm.$year,bk_fm.$d,bk_fm.$m,bk_fm.$y)
        else {
           put(tempf,"%d",code);
           fm_put(tempf,bk_fm);
           bks_cnt' := bks_cnt'+1;
        };
    };
    while new_bks /= <> do {
          code := code + 1;
          bk_fm:= hd new_bks;
          put(tempf,"%d",code);
          fm_put(tempf,bk_fm);
          put(logf,"New Book: %06d %s/%4d on %02d-%02d-%02d ^n",
                     code,bk_fm.$auth,bk_fm.$year,bk_fm.$d,bk_fm.$m,bk_fm.$d);
          bks_cnt' := bks_cnt'+1;
          new_bks := tl new_bks;
    };
    f_close(bkf);
    f_close(tempf);
    f_close(logf);
    tempf := f_open("temp","r+w");
    put(tempf,"%05d^n",bks_cnt');
    f_close(tempf);
    f_copy("temp","books");
END update_books
```

```
FUNCTION update_loans (lns_db: LoansDb, rds_db: ReadersDb, VAR rmv_rds: Id-list);
VAR lnf,logf: File;
    lns_list: Loan-list := db_list(lns_db);
    rds_list: Reader-list := db_list(rds_db);
    lns_cnt:  Nat0 := len lns_list;
    loan:     Loan;
    ln:       Code-list;
    resl:     Reserve-list;
    res:      Reserve;
    lost:     Code-list := <>;
    reader:   Reader;
    stays:    Bool := TRUE;
BEGIN
    lnf := f_open("loans","w");
    logf := f_open("loans.log","w");
    put(lnf,"%d %d^n",lns_cnt,day_no);
    while lns_list /= <> do {
        loan := hd lns_list;
        resl := loan.res;
        if loan.rec = 0 then {
           if loan.rd = NIL then {
              if resl = <> then
                 stays := FALSE
              else if resl[1].till /= NIL & resl[1].till < cur_date.no then {
                 loan.res := tl resl;
                 if loan.res = <> then
                    stays := FALSE
                 else {
                    loan.res[1].till := cur_date.no + 14;
                    put(logf,"Reserved: %06d for %05d^n",loan.code,loan.res[1].rd);
```

```
            };
        }
        else
            stays := FALSE;
    }
    else if loan.date+14 > cur_date.no then {
        loan.rec := 1;
        put(logf,"Recall Book: %06d from %05d ^n",loan.code,loan.rd);
    }
}
else if loan.rd = NIL then
    stays := FALSE
else if loan.rec<4 & loan.rec*30+14 <= cur_date.no - loan.date then {
    loan.rec := loan.rec+1;
    put(logf,"Recall Book(%d): %06d from %05d^n",loan.rec,loan.code,loan.rd);
}
else if loan.rec = 4 & loan.rec*30+14 > 200 then
    lost := <loan.code> || lost;
if stays then {
    put(lnf,"%d %d %d %d %d ",loan.code, loan.rd, loan.date, loan.rec, len resl);
    while resl /= <> do {
        res := hd resl;
        put(lnf,"%d %d %d ", res.date, res.rd,
                             if res.till = NIL then 0 else res.till);
        resl := tl resl;
    };
    put(lnf,"^n");
};
lns_list := tl lns_list;
};
while rds_list /= <> do {
    reader := hd rds_list;
    ln := reader.loan;
    while ln /= <> do {
        if ~is_element_of(hd ln,lost) then
            done;
        ln := tl ln;
    };
    if reader.loan /= <> & ln /= <> then
        rmv_rds := rmv_rds || <reader.id>;
    rds_list := tl rds_list;
};
f_close(lnf);
f_close(logf);
END update_loans

FUNCTION is_reserved_for(id: Id, resl: Reserve-list): Bool;
BEGIN
    while resl /= <> do {
        if id = (hd resl).rd then
            return(TRUE);
        resl := tl resl;
    };
    return(FALSE);
END is_reserved_for

FUNCTION message(line: Nat, kind: Message, message: Str);
BEGIN
    if kind = WARN then
```

```
        bell();
    w_move(line,2);
    w_put("^R%s",message);
    for i in {1 : w_spec('C') - st_len(message) - 2} do
        w_put(" ");
    w_put("^N");
END message

DIALOGUE remove_thing(what: What, VAR rmv_list: (Id | Code)-list, VAR del_ok: Bool);
VAR width:    Nat := 30;
    passwd:   Str;
    attempts: Nat0 := 0;
    ic:       Id | Code;
BEGIN
    state box: { assert(del_ok);
                 w_open(3,width, if what = READER then
                                    "^M Remove Reader ^N"
                                 else "^M Remove Book ^N");
                 message(3,NOTE,"");
               }                                    => pass;

    iap pass: { w_move(1,1);
                w_get(" Password:  ",passwd,8,noecho);
                message(3,NOTE,"");
              };
            : passwd = DEL_PASS                     => read;
            : attempts >= ATTEMPT_LIM,
              { message(3,WARN,"Imposter!");
                wait(2);
                del_ok := FALSE;
              }                                     => out;
            : TRUE, { attempts := attempts+1;
                      message(3,WARN,"Wrong!");
                    }                               => pass;

    iap read: { w_move(2,1);
                if what = READER then
                   w_get(" Reader Id: ",ic,5)
                else
                   w_get(" Book Code: ",ic,6);
              };
            : (if what = READER then db_find(rds_db,ic)
               else db_find(bks_db,ic)) /= NIL,
              { rmv_list := rmv_list || <ic>;
                message(3,NOTE,"Ok");
              }                                     => out;
            : ic = 0, message(3,NOTE,"Quited")      => out;
            : TRUE,   message(3,WARN,"Non-existant!") => read;

    state out: w_close(1)  => return;
END remove_thing

DIALOGUE insert_thing(what: What,
                      VAR new_list: (ReaderForm | BookForm)-list,
                      VAR ins_ok: Bool);
VAR width:    Nat := 30;
    passwd:   Str;
    attempts: Nat0 := 0;
    ok:       Bool;
```

```
    resp:     Char;
    rd_fm:    ReaderForm;
    bk_fm:    BookForm;
BEGIN
    state  box: { assert(ins_ok);
                  w_open(3,width, if what = READER then
                                    "^M New Reader ^N"
                                  else "^M New Book ^N");
                  message(3,NOTE,"");
                }                                    => pass;

    iap   pass: { w_move(1,1);
                  w_get(" Password:    ",passwd,8,noecho);
                  message(3,NOTE,"");
                };
                : passwd = INS_PASS                  => read;
                : attempts >= ATTEMPT_LIM,
                  { message(3,WARN,"Imposter!");
                    wait(2);
                    ins_ok := FALSE;
                  }                                  => out;
                : TRUE, { attempts := attempts+1;
                          message(3,WARN,"Wrong!");
                        }                            => pass;

    iap read: ok := if what = READER then
                        fm_new(rd_fm,"^M New Reader ^N")
                    else
                        fm_new(bk_fm,"^M New Book ^N");
                : ok & what = READER,
                  { new_list := new_list || <rd_fm>;
                    message(3,NOTE,"Registered");
                  }                                  => next;
                : ok & what = BOOK,
                  { new_list := new_list || <bk_fm>;
                    message(3,NOTE,"Recorded");
                  }                                  => next;
                : TRUE, message(3,NOTE,"Ignored")    => next;

    iap next: { w_move(2,1);
                w_put(" More [y/n]:      ");
                w_move(2,14);
                resp := keybd();
              };
              : resp = 'y' | resp = 'Y', w_put("yes")      => read;
              : resp = 'n' | resp = 'N', w_put("no ")      => out;
              : TRUE, message(3,WARN,"Yes or No please")   => next;

    state  out: w_close(1)                           => return;
END insert_thing

DIALOGUE issue_books (rds_db: ReadersDb, bks_db: BooksDb, lns_db: LoansDb);
VAR width: Nat := 30;
    id:    Id;
    code:  Code;
BEGIN
    state box: { w_open(3,width,"^M Issue ^N");
                 message(3,NOTE,"");
               }                                 => reader;
```

```
    iap reader: { w_move(1,1);
                  w_get(" Reader Id: ",id,5);
                };
              : id = 0                                  => out;
              : db_find(rds_db,id) = NIL,
                message(3,WARN,"No such reader")        => reader;
              : TRUE                                    => book;

    iap book: { w_move(2,1);
                w_get(" Book Code: ",code,6);
              };
            : code = 0                                  => out;
            : db_find(bks_db,code) = NIL,
              message(3,WARN,"No such book")            => book;
            : db_find(lns_db,code) /= NIL,
              message(3,WARN,"Is on loan")              => book;
            : TRUE                                      => issue;

    state issue: { db_insert(lns_db,mk-Loan(code,id,cur_date.no,0,<>));
                   message(3,NOTE,"Issued");
                 }                                      => book;
    state out: w_close(1)                               => return;
END issue_books

FUNCTION del_element(code: Code, codel: Code-list);
VAR head: Code := hd codel;
    tail: Code-list := tl codel;
    idx:  Nat := 2;
BEGIN
    if code /= head then
       while tail /= <> do {
           if code = hd tail then {
              codel[idx] := head;
              codel := tl codel;
              done;
           };
           tail := tl tail;
           idx := idx+1;
       }
    else
       codel := tl codel;
END del_element

DIALOGUE discharge_books(rds_db: ReadersDb, lns_db: LoansDb);
VAR width:  Nat := 30;
    code:   Code;
    loan:   [Loan];
    reader: Reader;
BEGIN
    state box: { w_open(3,width,"^M Discharge ^N");
                 message(2,NOTE,"");
                 message(3,NOTE,"");
               }                                        => book;

    iap book: { w_move(1,1);
                w_get(" Book Code: ",code,6);
              };
            : code = 0                                  => out;
            : (loan := db_find(lns_db,code)) = NIL | loan.rd    NIL,
```

```
            message(2,WARN,"Is not on loan")   => book;
          : TRUE                               => disch;

    state disch: { reader := db_find(rds_db,loan.rd);
                   del_element(code,reader.loan);
                   loan.rd := NIL;
                   message(2,NOTE,"Discharged");
                   mac {
                     loan.rec>0  => message(3,WARN,"Goes to RECALLED shelf");
                     loan.res=<> => { message(3,NOTE,"Goes to shelves");
                                      db_delete(lns_db,code);
                                    };
                     TRUE        => { message(3,WARN,"Goes to RESERVE shelf");
                                      loan.res[1].till := cur_date.no;
                                    };
                   };
                 }                             => book;

    state out: w_close(1)                      => return;
END discharge_books

DIALOGUE renew_books(lns_db: LoansDb);
VAR width:  Nat := 30;
    code:   Code;
    loan:   [Loan];
    reader: Reader;
BEGIN
    state box: { w_open(2,width,"^M Renew ^N");
                 message(2,NOTE,"");
               }                               => book;

    iap book: { w_move(1,1);
                w_get(" Book Code: ",code,6);
              };
            : code = 0                         => out;
            : (loan := db_find(lns_db,code)) = NIL | loan.rd = NIL,
              message(2,WARN,"Is not on loan") => book;
            : loan.rec > 0,
              message(2,WARN,"Recalled - can't renew") => book;
            : loan.res /= <>,
              message(2,WARN,"Reserved - can't renew") => book;
            : TRUE                             => renew;

    state renew: loan.date := cur_date.no      => out;
    state out: w_close(1)                      => return;
END renew_books

DIALOGUE reserve_books(rds_db: ReadersDb, lns_db: LoansDb);
VAR width: Nat := 30;
    id:    Id;
    code:  Code;
    loan:  [Loan];
BEGIN
    state box: { w_open(3,width,"^M Reserve ^N");
                 message(3,NOTE,"");
               }                               => reader;

    iap reader: { w_move(1,1);
                  w_get(" Reader Id: ",id,5);
                };
```

```
            : id = 0                                  => out;
            : db_find(rds_db,id) = NIL,
              message(3,WARN,"No such reader")        => reader;
            : TRUE                                    => book;

    iap book: { w_move(2,1);
                w_get(" Book Code: ",code,6);
              };
            : code = 0                                          => out;
            : (loan := db_find(lns_db,code)) = NIL,
              message(3,WARN,"Is not on loan")                  => book;
            : loan.rd = id,
              message(3,WARN,"Reader has the book")             => book;
            : is_reserved_for(id,loan.res),
              message(3,WARN,"Already reserved for reader")     => book;
            : TRUE                                              => reserve;

    state reserve: { loan.res := loan.res || <mk-Reserve(cur_date.no,id,NIL)>;
                     message(3,NOTE,"Reserved");
                   }                                    => book;
    state out: w_close(1)                               => return;
END reserve_books

CLUSTER dial_box {
          title:Const: Str
          { 'field' fld:Const: Str ',' fid:Ident: (Str | Int | Real)
                                   ':' fsz:Const: Nat
                                   ',' 'empty' emp:Const: (Str | Int | Real)
                    [ '=>' 'commands']co ';'
          }+fr
          { 'command' comnd:Const: Str '=>' action:Statm
                                         ['=>' 'fields']fo ';'

          }+cr
        };
VAR flen: Nat0;
    lins: Nat := fr+3;
    cols: Nat;
    sum:  Nat := 2;
    max_len, max_siz: Nat := 1;
    com_pos: array[cr+1] Nat;
    id:   Str | Int | Real;
    ch:   Char;
    op:   Nat;
BEGIN
    for i in (1:fr) do {
        if (flen := st_len(fld[i])) > max_len then
           max_len := flen;
        if fsz[i] > max_siz then
           max_siz := fsz[i];
    };
    cols := max_len+max_siz;
    for i in (1:cr) do {
        com_pos[i] := sum;
        sum := com_pos[i]+st_len(comnd[i])+2;
    };
    if sum > cols then
       cols := sum;

    w_open(lins,cols,title);
```

```
for i in {1:fr} do
    w_put("%s^n",fld[i]);
for i in {1:cr} do {
    w_move(fr+1,com_pos[i]+1);
    w_put("^R%s^N",comnd[i]);
};
message(lins,NOTE,"");

while TRUE do {
    for i in {1:fr} do {
        w_move(i,max_len+1);
        w_get(id,fsz[i]);
        fid[i] := id;
        if fid[i] /= emp[i] & co[i] = 1 then
           done;
    };
    op := 1;
    while TRUE do {
        w_move(fr+1,com_pos[op]+1);
        w_put("^M%s^N",comnd[op]);
        w_move(fr+1,com_pos[op]);

        cases (ch := keybd()) {
          'F1' => { w_put(" ^R%s^N",comnd[op]);
                    op := if op = 1 then cr else op-1;
                  };
          'F2' => { w_put(" ^R%s^N",comnd[op]);
                    op := if op = cr then 1 else op+1;
                  };
          '^r' => { action[op];
                    if fo[op] = 1 then {
                       w_put(" ^R%s^N",comnd[op]);
                       done;
                    };
                  };
          TRUE => bell();
        };
    };
};

on_exit do
    w_close(1);
END dial_box

FUNCTION sort_by_pos(items: (Reader | Book)-list);
VAR swap:   Bool := TRUE;
    length: Nat0 := len items;
    temp:   (Reader | Book);
BEGIN
    while swap do {
        swap := FALSE;
        for i in {1:length-1} do
            if items[i].pos > items[i+1].pos then {
               temp := items[i];
               items[i] := items[i+1];
               items[i+1] := temp;
               swap := TRUE;
            };
    };
```

```
END sort_by_pos

FUNCTION find_readers_dial(rds_db: ReadersDb);
VAR id:         Id;
    sname:      Name;
    readers:    ReaderForm-list := <>;
    count:      Nat0 := 0;

    FUNCTION find_readers(id: Id, sname: Name): ReaderForm-list;
    VAR rdf:        File;
        rds_cnt:    Nat0;
        rd:         [Reader];
        rds_list':  Reader-list;
        rds:        Reader-list := <>;
        rd_fm:      ReaderForm;
        rd_fms:     ReaderForm-list := <>;
        valid:      Bool;
        count:      Nat0;
        code:       Code;
        from:       Position;

     BEGIN
         rdf := f_open("readers","r");
         get(rdf,rds_cnt);
         f_zap(rdf);
         if id /= 0 then {
            if (rd := db_find(rds_db,id)) /= NIL then {
               for i in {1:rd.pos} do
                   f_zap(rdf);
               get(rdf,id,valid,count);
               for i in {1:count} do
                   get(rdf,code);
               fm_get(rdf,rd_fm);
               f_close(rdf);
               rd_fms := <rd_fm>;
            };
         }
         else {
            rds_list' := rds_list;
            while rds_list' /= <> do {
                  if st_sub(sname,(hd rds_list').name) then
                     rds := rds || <hd rds_list'>;
                  rds_list' := tl rds_list';
            };
            if rds /= <> then {
               sort_by_pos(rds);
               from := 0;
               while rds /= <> do {
                   for i in {from:(hd rds).pos-1} do
                       f_zap(rdf);
                   get(rdf,id,valid,count);
                   for i in {1:count} do
                       get(rdf,code);
                   fm_get(rdf,rd_fm);
                   rd_fms := rd_fms || <rd_fm>;
                   from := (hd rds).pos+1;
                   rds := tl rds;
               };
            };
```

```
        };
        f_close(rdf);
        return(rd_fms);
    END find_readers
BEGIN
    dial_box {
        "^M Find Reader ^N"
        field " Id Number: ",  id: 5,  empty 0 => commands;
        field " Surname: ", sname: 20, empty "" ;
        command " FIND " => { readers := find_readers(id,sname);
                              count := 1;
                              cases len readers {
                                0    => message(5,WARN,"Can't find reader");
                                1    => fm_view(hd readers,"");
                                TRUE => message(5,NOTE,
                                                st_app(st_num(len readers)," hits"));
                              };
                            };
        command " NEXT " => { if readers = <> then
                                 message(5,WARN,"No reader found yet")
                              else {
                                 fm_view(hd readers,
                                         st_app(st_app("^M Item £",st_num(count)),
                                                " ^N"));
                                 count := count+1;
                                 readers := tl readers;
                                 message(5,NOTE,
                                          st_app(st_num(len readers)," remaining"));
                              };
        command " BACK " => message(5,NOTE,"") => fields;
        command " QUIT " => exit;
    };
END find_readers_dial

FUNCTION find_books_dial(bks_db: BooksDb);
VAR code:  Code;
    auth:  Author;
    title: Title;
    books: BookForm-list;
    count: Nat0 := 0;

    FUNCTION find_books(code: Id, auth: Author, title: Title): BookForm-list;
    VAR bkf:       File;
        rds_cnt:   Nat0;
        bk:        [Book];
        bks_list': Book-list;
        bks:       Book-list := <>;
        bk_fm:     BookForm;
        bk_fms:    BookForm-list := <>;
        from:      Position;
    BEGIN
        bkf := f_open("books","r");
        get(bkf,rds_cnt);
        f_zap(bkf);
        if code /= 0 then {
           if (bk := db_find(bks_db,code)) /= NIL then {
              for i in {1:bk.pos} do
                  f_zap(bkf);
```

```
            get(bkf,code);
            fm_get(bkf,bk_fm);
            f_close(bkf);
            bk_fms := <bk_fm>;
        };
    }
    else {
        bks_list' := bks_list;
        while bks_list' /= <> do {
              if st_sub(auth,(hd bks_list').auth) &
                 st_sub(title,(hd bks_list').titl) then
                 bks := bks || <hd bks_list'>;
              bks_list' := tl bks_list';
        };
        if bks /= <> then {
            sort_by_pos(bks);
            from := 0;
            while bks /= <> do {
                for i in {from:(hd bks).pos-1} do
                    f_zap(bkf);
                get(bkf,code);
                fm_get(bkf,bk_fm);
                bk_fms := bk_fms || <bk_fm>;
                from := (hd bks).pos+1;
                bks := tl bks;
            };
        };
    };
    f_close(bkf);
    return(bk_fms);
END find_books
BEGIN
    dial_box {
        "^M Find Book ^N"
        field " Code: ",    code: 6,  empty 0 => commands;
        field " Author: ",  auth: 20, empty "";
        field " Title: ",  title: 25, empty "";
        command " FIND " => { books := find_books(code,auth,title);
                              count := 1;
                              cases len books {
                                0    => message(6,WARN,"Can't find book");
                                1    => fm_view(hd books,"");
                                TRUE => message(6,NOTE,
                                                st_app(st_num(len books)," hits"));
                              };
                            };
        command " NEXT " => { if books = <> then
                                 message(6,WARN,"No book found yet")
                              else {
                                 fm_view(hd books,
                                         st_app(st_app("^M Item £",st_num(count)),
                                                " ^N"));
                                 count := count+1;
                                 books := tl books;
                                 message(6,NOTE,st_app(st_num(len books),"remaining"));
                              };
                            };
        command " BACK " => message(6,NOTE,"") => fields;
        command " QUIT " => exit;
```

```
    };
END find_books_dial

FUNCTION start_up ();
BEGIN
    move(24,2);
    put("^MPlease Wait^N");
    init_readers();
    init_books();
    rds_list := db_list(rds_db);
    bks_list := db_list(bks_db);
    day_no := init_loans();
    cur_date := mk-TopDate(0,time('Y'),time('M'),time('D'));
    cur_date.no := cur_date.y*365 + cur_date.m*30 + cur_date.d;
    move(24,2);
    put("^RDAY: %d     ^N",day_no);
    move(24,48);
    put("^RUP: at %02d:%02d:%02d, on %02d %3s %4d",
        time('h'),time('m'),time('s'),
        cur_date.d,MONTHS[cur_date.m],1900+cur_date.y);
END start_up

FUNCTION counter_desk_menu ();
BEGIN
    menu {
       "^M Counter Desk ^N"
       "Issue"      => issue_books(rds_db,bks_db,lns_db);
       "Discharge"  => discharge_books(rds_db,lns_db);
       "Renew"      => renew_books(lns_db);
       "Reserve"    => reserve_books(rds_db,lns_db);
       "Quit"       => exit;
       TRUE         => exit;
    };
END counter_desk_menu

FUNCTION reader_menu ();
BEGIN
    menu {
       "^M Reader ^N"
       "New Reader",    constraint ins_ok => insert_thing(READER,new_rds,ins_ok);
       "Remove Reader", constraint del_ok => remove_thing(READER,rmv_rds,del_ok);
       "Find Reader"                      => find_readers_dial(rds_db);
       "Quit"                             => exit;
       TRUE                               => exit;
    };
END reader_menu

FUNCTION book_menu ();
BEGIN
    menu {
       "^M Book ^N"
       "New Book",    constraint ins_ok => insert_thing(BOOK,new_bks,ins_ok);
       "Remove Book", constraint del_ok => remove_thing(BOOK,rmv_bks,del_ok);
       "Find Book"                      => find_books_dial(bks_db);
       "Quit"                           => exit;
       TRUE                             => exit;
    };
END book_menu
```

```
FUNCTION report_menu ();
   FUNCTION prepare_stock_report ();
   VAR bkf, stkf: File;
       bks_cnt:   Nat0;
       code:      Code;
       bk_fm:     BookForm;
   BEGIN
       bkf := f_open("books","r");
       stkf := f_open("stock.log","w");
       get(bkf,bks_cnt);
       f_zap(bkf);
       for i in {1:bks_cnt} do {
           get(bkf,code);
           fm_get(bkf,bk_fm);
           put(stkf,"%06d %s %s --- L%03d.%03d /%d/ ISBN 0-%03d-%05d-%d ^n",
                    code, bk_fm.$auth, bk_fm.$i, bk_fm.$cl, bk_fm.$cr,
                    bk_fm.$year, bk_fm.$s1, bk_fm.$s2, bk_fm.$s3);
           put(stkf,"       %s ^n       %s, %02d-%02d-%02d -- %d(%d) ^n",
                    bk_fm.$titl, bk_fm.$pub, bk_fm.$d, bk_fm.$m, bk_fm.$y,
                    bk_fm.$vol, bk_fm.$edtn);
       };
       f_close(bkf);
       f_close(stkf);
   END prepare_stock_report
BEGIN
    menu {
       "^M Reports ^N"
       "Readers" => w_text(10,60,"^M Readers Log ^N","readers.log");
       "Books"   => w_text(10,60,"^M Books Log ^N","books.log");
       "Loans"   => w_text(10,60,"^M Loans Log ^N","loans.log");
       "Entire Stock"
                 => { if ~stock_rep_ready then {
                          w_open(2,27,"");
                          w_put("^BPlease Wait^N^nwhile I prepare the report");
                          prepare_stock_report();
                          stock_rep_ready := TRUE;
                          w_close(1);
                      };
                      w_text(10,60,"^M Stock Log ^N","stock.log");
                    };
       "Quit"    => exit;
       TRUE      => exit;
    };
END report_menu

FUNCTION shut_down (started: Bool);
BEGIN
    if started then {
       move(24,2);
       put("^MPlease Wait^N");
       update_loans(lns_db,rds_db,rmv_rds);
       update_readers(rds_db,new_rds,rmv_rds);
       update_books(new_bks,rmv_bks);
    };
END shut_down

BEGIN /* design */
    init_scr();
    put("^R%26sL I B R A R Y     S Y S T E M%26s^N","","");
```

```
        move(24,1);
        put("^R%80s^N","");
        move(3,2);

        menu (
           ""
           "Start Up",      constraint ~started => {start_up(); started := TRUE};
           "Counter Desk", constraint  started => counter_desk_menu();
           "Reader",        constraint  started => reader_menu();
           "Book",          constraint  started => book_menu();
           "Reports",       constraint  started => report_menu();
           "Shut Down"                          => { shut_down(started);
                                                     exit;
                                                   };
    };
    tini_scr();
END library_system
```